# Bridging the gap between OpenMP 4.0 and native runtime systems for the fast multipole method

Emmanuel Agullo, Olivier Aumage, Berenger Bramas, Olivier Coulaud, Samuel Pitoiset


# Bridging the gap between OpenMP 4.0 and native runtime systems for the fast multipole method

**Emmanuel Agullo, Olivier Aumage, Berenger Bramas, Olivier Coulaud, Samuel Pitoiset**

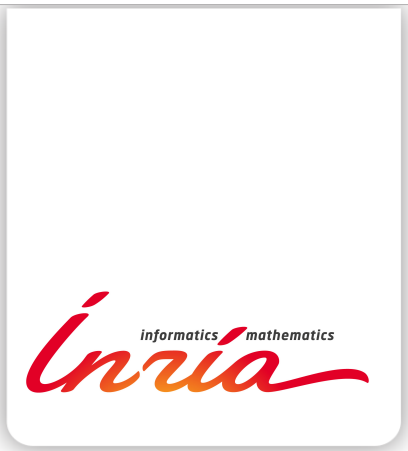
**RESEARCH REPORT**

**N° 8953**

May 2016

Project-Teams HiePACS and Storm

ISSN 0249-6399 ISRN INRIA/RR--8953--FR+ENG




# Bridging the gap between OpenMP 4.0 and native runtime systems for the fast multipole method

Emmanuel Agullo, Olivier Aumage, Berenger Bramas, Olivier Coulaud, Samuel Pitoiset

Project-Teams HiePACS and Storm

Research Report n° 8953 — May 2016 — 49 pages

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

**Abstract:** With the advent of complex modern architectures, the low-level paradigms long considered sufficient to build High Performance Computing (HPC) numerical codes have met their limits. Achieving efficiency, ensuring portability, while preserving programming tractability on such hardware prompted the HPC community to design new, higher level paradigms. The successful ports of fully-featured numerical libraries on several recent runtime system proposals have shown, indeed, the benefit of task-based parallelism models in terms of performance portability on complex platforms. However, the common weakness of these projects is to deeply tie applications to specific expert-only runtime system APIs. The OpenMP specification, which aims at providing a common parallel programming means for shared-memory platforms, appears as a good candidate to address this issue thanks to the latest task-based constructs introduced as part of its revision 4.0.
The goal of this paper is to assess the effectiveness and limits of this support for designing a high-performance numerical library. We illustrate our discussion with the ScalFMM library, which implements state-of-the-art fast multipole method (FMM) algorithms, that we have deeply re-designed with respect to the most advanced features provided by OpenMP 4. We show that OpenMP 4 allows for significant performance improvements over previous OpenMP revisions on recent multicore processors. We furthermore propose extensions to the OpenMP 4 standard and show how they can enhance FMM performance. To assess our statement, we have implemented this support within the Klang-OMP source-to-source compiler that translates OpenMP directives into calls to the StarPU task-based runtime system. This study shows that we can take advantage of the advanced capabilities of a fully-featured runtime system without resorting to a specific, native runtime port, hence bridging the gap between the OpenMP standard and the very high performance that was so far reserved to expert-only runtime system APIs.

**Key-words:** runtime system, parallel programming model, compiler, OpenMP 4.0, OpenMP 4.X, priority, commutativity, multicore architecture

# Combler l'écart de performance entre OpenMP 4.0 et les moteurs d'exécution pour la méthode des multipoles rapide

**Résumé :** Avec l'arrivée des architectures modernes complexes, les paradigmes de parallélisation de bas niveau, longtemps considérés comme suffisant pour développer des codes numériques efficaces, ont montré leurs limites. Obtenir de l'efficacité et assurer la portabilité tout en maintenant une bonne flexibilité de programmation sur de telles architectures ont incité la communauté du calcul haute performance (HPC) à concevoir de nouveaux paradigmes de plus haut niveau. Les portages réussis de bibliothèques numériques sur plusieurs moteurs exécution récentos ont montré l'avantage des modèles de parallélisme à base de tâche en ce qui concerne la portabilité et la performance sur ces plateformes complexes. Cependant, la faiblesse de tous ces projets est de fortement coupler les applications aux experts des API des moteurs d'exécution. La spécification d'OpenMP, qui vise à fournir un modèle de programmation parallèle unique pour les plates-formes à mémoire partagée, semble être un bon candidat pour résoudre ce problème. Notamment, en raison des améliorations apportées à l'expressivité du modèle en tâches présentées dans sa révision 4.0.

Le but de ce papier est d'évaluer l'efficacité et les limites de ce modèle pour concevoir une bibliothèque numérique performante. Nous illustrons notre discussion avec la bibliothèque ScalFMM, qui implémente les algorithmes les plus récents de la méthode des multipôles rapide (FMM). Nous avons finement adapté ces derniers pour prendre en compte les caractéristiques les plus avancées fournies par OpenMP 4. Nous montrons qu'OpenMP 4 donne de meilleures performances par rapport aux versions précédentes d'OpenMP pour les processeurs multi-coeurs récents. De plus, nous proposons des extensions au standard d'OpenMP 4 et nous montrons comment elles peuvent améliorer la performance de la FMM. Pour évaluer notre propos, nous avons mis en oeuvre ces extensions dans le compilateur source-à-source Klang-OMP qui traduit les directives OpenMP en des appels au moteur d'exécution à base de tâches StarPU. Cette étude montre que nous pouvons tirer profit des capacités avancées du moteur d'exécution sans devoir recourir à un portage sur l'API spécifique de celui-ci. Par conséquent, on comble le fossé entre le standard OpenMP et l'approche très performante par moteur d'exécution qui est de loin réservée au seul expert son API.

**Mots-clés :** moteur d'exécution, modèle de programmation parallèle , compilateur, OpenMP 4.0, OpenMP 4.X, priorité, commutativité, architecture multicore

# Contents

# List of Algorithms

# 1 Introduction

The advent of modern computing architectures with large number of cores puts a strong pressure on parallel programming paradigms. The low-abstraction, thread-based paradigms reach their limits, due to the difficulty to handle the resulting management and synchronization complexity for a programmer. As a consequence, the High Performance Computing (HPC) community is investigating the design of new, higher level programming paradigms. Among such paradigms, task-based parallelism models have been proposed and implemented in several robust runtime systems. The successful ports of popular numerical libraries on top of these runtimes have demonstrated their effectiveness and benefit in terms of performance portability on complex platforms. However, each task-based runtime system comes with its own, dedicated application programming interface, which may differ significantly from the API of other task-based runtimes. The result is a babel tower-like scheme where choosing a given runtime to port an application deeply ties the application to it. A major objective of the OpenMP specification is to define a common, abstract programming method for shared-memory parallel platforms. OpenMP therefore appears as a prominent candidate to address the issue of the dedicated task-based runtime systems API profusion, since the introduction of tasks (OpenMP rev 3.x) and more recently dependent tasks (OpenMP rev 4.0) in the specification.

Pondering the use of an abstract layer such as OpenMP as the programming model for a parallel HPC application, involves questioning its benefits and performance trade-offs beyond the mere gain in portability. The goal of this paper is to explore these benefits and trade-offs for a Fast Multipole Methods (FMM) library, ScalFMM, aiming at computing pair-wise particles interactions and whose an overview is presented in Section 3. The main interest of this application is to generate very different workloads and to expose widely differing behaviours depending on the input datasets. The study is conducted both with the GNU GCC compiler, which targets its own LibGOMP runtime system, and with our Klang compiler, for which we selected the StarPU runtime target. The Gcc/LibGomp couple is one of the reference implementation of the OpenMP specification, and offers an efficient, lightweight task scheduler. The Klang/-StarPU couple also provides the compliance with the OpenMP `task` directive, while offering the opportunity to experiment with accessing StarPU additional features such as advanced scheduling or data management policies.

We show that OpenMP 4 allows for significant performance improvements over previous OpenMP revisions on recent multicore processors. We also identify situations where this abstraction may prevent leveraging useful runtime system features. Using pinpointed *addenda* to the OpenMP `task` directive, we show that opportunities for extensions exist to give additional driving hints to the underlying runtime system, and take advantage of its advanced capabilities when they are relevant for a given case, without resorting to a specific, native runtime port of the application.

The contributions of this study are the following:

- delivering a new design of the ScalFMM library with enhanced performance on multicore architectures, compatible with the OpenMP 4 standard;
- evaluating the behaviour of a reference lightweight (LibGOMP) and a fully-featured (StarPU) back-end runtime system to OpenMP compilers;
- proposing optional extensions to the OpenMP 4 standard and assessing how they can speed-up a target state-of-the-art scientific library.

All in all, we show that we can bridge the productivity versus performance gap between OpenMP and native runtime systems, ensuring the performance of a code tailored for a native runtime

system with the compactness and elegance of OpenMP.

The rest of the paper is organized as follows. Section 2 presents both the abstraction effort put in designing the OpenMP specification, and the concurrent enterprise of the community to design new powerful runtime systems to efficiently execute parallel applications on high performance platforms; it also exposes related work investigating their combination. We propose a brief overview of the FMM in Section 3 to make the paper self-contained, together with guidelines for the reader in a hurry to proceed to the FMM parallelization schemes proposed in Section 4. Thanks to new data structures, we propose extremely compact schemes, based on either OpenMP directives or StarPU routines. We implemented them in the ScalFMM library for the purpose of the present study. Section 5 presents a performance analysis, before Section 6 concludes this paper and presents future research directions.

# 2 Background

## 2.1 OpenMP

The OpenMP ARB (Architecture Review Board) published the first revisions of the OpenMP specification [21] by the end of the nineties and the beginning of years 2000, with the aim to provide an abstract, portable, programming model and make parallel programming on shared-memory machines a straightforward and user-friendly process. It provided basic constructs to define parallel regions in a fork-join fashion, and build parallel loops as well as parallel sections on top of such regions. At this time, with the exception of a few programming environments such as Cilk [15], programming shared-memory machines necessitated tricky, error-prone manipulations of thread objects provided by the various POSIX threads compliant libraries or custom multithreading library flavors. The specification of OpenMP has since been revised and enriched several times, especially introducing constructs for independent-tasks programming model (rev. 3.0/3.1 [8]), and dependent tasks (rev. 4.0 [21]), besides many other features.

## 2.2 Task-based runtime systems

The job of managing the execution and of mapping the parallelism of an application onto computing units has been the subject of numerous works. It led to the proposal of many runtime systems to implement scheduling algorithms based on theoretical scheduling researches. Among runtime systems, and especially since the emergence of multicore processors, task-based execution models have become popular parallel application foundations. As the number of cores increases in processors, and may greatly vary from platform to platform, the ability of task execution models to flexibly map computations on available resources is a key reason of their success [7, 10, 11, 15, 16, 20]. However, each task-based runtime system comes with its dedicated programming interface. Thus, an application ported on a given runtime will necessitate additional programming and refactoring code to run on another runtime, resulting in reduced portability and preventing comparisons between different runtime systems.

## 2.3 OpenMP/ runtime systems relationships

All OpenMP-compliant compilers underneath rely on a runtime system to support the parallel execution of the compiled code. However, reference implementations tend to privilege lightweight engines in order to limit the overhead that advanced policies could induce. For instance, GCC relies on the LibGOMP lightweight runtime system, and the Gcc/LibGomp couple got its success in being one of the reference implementation of the OpenMP specification, because

this design allows users to achieve competitive performance, for embarrassingly or moderately complex parallel cases, with an extremely high programming productivity.

Those users, instead, who need to achieve even higher performance on test cases exhibiting complex parallelism structure, and endeavour to exploit the most of modern platforms' power, the option to resort to fully-featured task-based runtime systems — designed for that very purpose — is impeded by productivity issues. The learning curve may be very steep and comes with the additional drawback of tying the application to a specific tool. As a result, several proposals have been made to help combine the abstraction of the OpenMP programming model with the benefits of fully-featured task-based runtime systems. These frameworks have in common to rely on a source-to-source OpenMP compiler responsible for translating abstract OpenMP constructs into calls to a runtime system dedicated API. The ROSE compiler [18] developed at LLNL is designed to support multiple back-end runtime libraries. The OpenUH compiler [4], developed at the University of Houston is based on the Open64 compiler framework and provides both source-to-source compilation as well as binary generation. The Mercurium [11, 14] source-to-source compiler developed at Barcelona Supercomputing Center targets the Nanos++ runtime system, and supports OpenMP compliance as well as OmpSs extensions to the OpenMP specification. In particular, Mercurium has been used as a prototype for designing the OpenMP tasking model during the latest specification revision processes. Commercial compilers also come with their own runtime system, such as the IBM XL compiler [23], or the Intel compiler (runtime ABI shared with LLVM [3]).

These proposals are made possible by the abstract, implementation independent architecture of OpenMP. The language defines concepts such as *device*, *thread*, *thread team*, *implicit task* (a task implicitly arising from encountering an OpenMP `parallel` construct) and *explicit task* (a task arising from encountering an OpenMP `task` construct), to name the most useful here. Conceptually, a host device may launch teams of threads, where each thread can run implicit and explicit tasks. Tasks may create new nested tasks, new nested threads teams and so on (see OpenMP's specification [22]). OpenMP also defines rules to create corresponding objects, as well as rules enabling to collapse such objects. This enables different possible implementation strategies, accommodating different hardware capabilities (as an extreme case, these objects can be entirely collapsed to a sequential execution). In particular, implicit tasks are often collapsed with their corresponding thread, in which case no dedicated structure are allocated for them by the runtime system following this implementation strategy. This is for instance the case for Gcc/LibGomp and for Nanos++.

In this paper, we rely on the Klang C/C++ source-to-source OpenMP compiler together with the StarPU runtime system [7]. The Klang compiler is based on the LLVM framework and on Intel's Clang-OMP front-end. It translates OpenMP directives into calls to task-based runtime system APIs such as the StarPU runtime. Klang supports legacy fork-join OpenMP constructs such as `parallel` regions, `parallel for` loops and sections. It also supports independent tasks as defined by the OpenMP specification revision 3.1 as well as dependent tasks introduced with OpenMP 4.0. From an implementation point of view, all *explicit task regions* (arising from both dependent and independent tasks in OpenMP terminology) are directly mapped on StarPU tasks. Legacy fork-join constructs are implemented by mapping their *implicit task regions* on StarPU tasks. All StarPU task scheduling algorithms (priority, work-stealing, ...) available to "native" StarPU applications are available as well for OpenMP programs running on top of StarPU. We further discuss these implementation considerations in Section 4.3

# 3 Brief overview of the FMM

The FMM is a hierarchical algorithm originally introduced in [17]. It aims to reduce the quadratic complexity of pair-wise interactions to a linear or a linearithmic complexity. The FMM is now used in a large range of applications such as vertex methods, boundary element methods (BEM) or radial basis functions. In order to make the paper self-contained, we present a brief overview of the FMM algorithm, but **the reader in a hurry may proceed to Section 4 knowing that the issue we address in Section 4 will be to parallelize Algorithm 1 with a concise OpenMP code, in order to exploit the potential parallelism it provides as expressed in the example DAG in Figure 3b**.

The key-point of the FMM algorithm is to approximate the far-field - the interactions between far-apart particles - while maintaining a desired accuracy, exploiting the property that the underlying mathematical kernel decays with the distance between particles. While the interactions between close particles still remain computed with a direct Particle to Particle (P2P) method, the far-field is processed using a tree-based algorithm instead. A recursive subdivision of the space is performed in a preprocessing symbolic step (see Figure 1). This recursive subdivision is usually represented with a hierarchical tree data structure and we call the height of the tree $h$ the number of recursions. The type (quadtree, octree, ...) of the tree is related to the dimension of

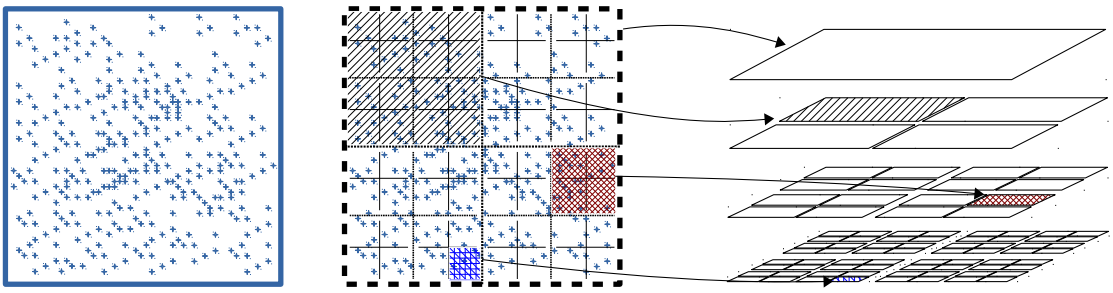


Figure 1: 2D space decomposition (Quadtree). Grid view and hierarchical view.

the problem. However, in the current study we use the term *octree* in a generic manner to refer to the FMM tree for any dimension. Figure 1 is an example of an octree showing the relationship between the spatial decomposition and the data structure, and we see that each cell represents its descendants composed of its children and sub-children. The multipole (M) of a given cell represents the contribution of its descendants. On the other hand, the local part (L) of a cell $c$ represents some contributions that will be applied to the descendants of $c$. These local contributions in a cell $c$ come from the potential of particles/cells that are not included by $c$. Relying on those recursive data structures, the FMM algorithm proceeds in four steps as described in Figure 2, namely the upward pass, the transfer pass, the direct pass and the downward pass.

In an upward pass of the FMM, the physical values of the particles are aggregated from bottom to top using the Particle to Multipole ($P2M$) and Multipole to Multipole ($M2M$) operators. After this operation, each cell hosts the contributions of its descendants. In the transfer pass, the Multipole to Local ($M2L$) operator is applied between each cell and its corresponding interaction list at all levels. The interaction list for a given cell $c$ at level $l$ is composed by the children of the neighbors of $c$'s parent that are not direct neighbors/adjacent to $c$. After the transfer pass, the local part of all the cells are filled with contributions. The downward pass aims to apply these contributions to the particles. In this pass, the local contributions are propagated from top to bottom with the Local to Local ($L2L$) operator, and applied to the particles with the Local to Particle ($L2P$) operator. After these far-field operations, the particles have received

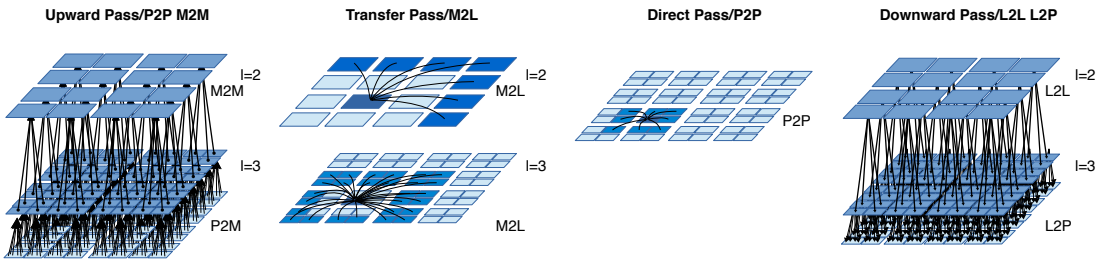



Figure 2: Successive steps of the FMM algorithm; upward pass (left), transfer pass and direct step (center), and downward pass (right).

their respective far contributions.

Algorithm 1 shows the resulting sequential pseudo-code (for a matter of conciseness, we provide the detailed code of the M2M operator only). The dependencies between the operations

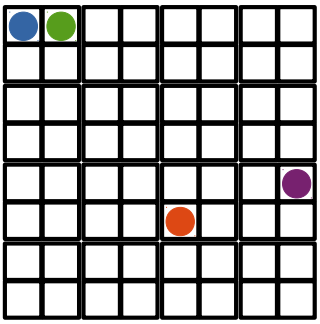
**Algorithm 1:** FMM Sequential Algorithm

```
1  function FMM(tree, kernel)
2      // Near-field
3      P2P(tree, kernel);
4      // Far-field
5      P2M(tree, kernel);
6      for l = tree.height-2 → 2 do
7          M2M(tree, kernel, l);
8      for l = 2 → tree.height-2 do
9          M2L(tree, kernel, l);
10         L2L(tree, kernel, l);
11     M2L(tree, kernel, tree.height-1);
12     L2P(tree, kernel);
13 function M2M(tree, kernel, level)
14     foreach cell cl in tree.cells[level] do
15         kernel.M2M(cl.multipole, tree.getChildren(cl.mindex, level).multipole);
```

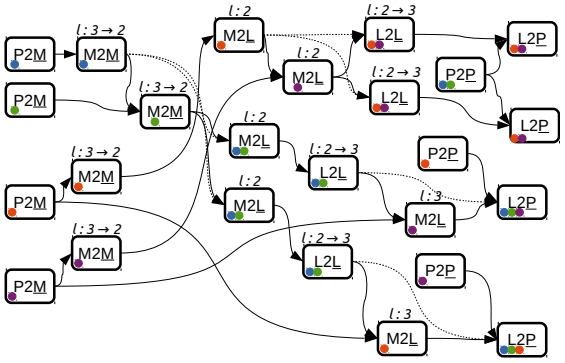
occurring within this algorithm can be encoded with a Directed Acyclic Graph (DAG) where vertices represent operators and edges dependencies between them. Figure 3b for instance represents the DAG of the FMM associated with the tiny grid provided in Figure 3a.

# 4 Bridging the performance gap between OpenMP-based and native runtime systems-based FMM

In a previous study [5], we showed that elaborated and complex FMM parallelization schemes based on OpenMP 3.0 were not competitive against task-based designs natively exploiting runtime systems. On the contrary, we now propose concise OpenMP-based FMM implementations compliant not only with the OpenMP standard but also with the original idea of the OpenMP ARB to make parallel programming on shared-memory machines a user-friendly process. The resulting codes will then be used in Section 5 to assess whether the new revision of the OpenMP standard allows for achieving both high performance and programming productivity.



(a) Particles layout, tree height $h = 4$.




(b) Corresponding FMM DAG. The colored circles represent the contributions, such that the L2P contains the far-field potential to apply on the leaves.

Figure 3: Example: Interaction between 4 particles.

## 4.1 Fork-Join schemes

A straightforward FMM parallelization scheme consists in performing a level by level parallelization of each inner loop with a *#pragma omp parallel for* directive. Such a parallelization can be implemented with the sequential algorithm (Algorithm 1) within which the main loop of each operator implementation is prepended with this directive [2, 1, 24]. Algorithm 2 shows how the M2M operator is readily adapted. We name *fj-omp#for-dyn* this fork-join (*fj*) approach based on the *#pragma omp parallel for* dynamic loop scheduling directive.

**Algorithm 2:** *fj-omp#for-dyn*

```
1 function M2M(tree, kernel, level)
2     #pragma omp parallel for schedule(dynamic, 10)
3         foreach cell cl in tree.cells[level] do
4             kernel.M2M(cl.multipole, tree.getChildren(cl.mindex, level).multipole);
5         // Implicit barrier from omp parallel
```

We consider a second variant based on static loop scheduling, where we aim at assigning the same amount of work to each thread. We use the number of interactions per elements as the balancing metric, and a greedy pre-processing to find the correct interval. We name *fj-omp#for-sta* this static variant in the sequel whose M2M code is provided in Algorithm 3.

**Algorithm 3:** *fj-omp#for-sta*

```
1 function M2M(tree, kernel, level)
2     #pragma omp parallel
3         s = thread_interval[level][omp_get_threadnum()].start;
4         e = thread_interval[level][omp_get_threadnum()].end;
5         foreach cell cl in tree.cells[level] from s to e do
6             kernel.M2M(cl.multipole, tree.getChildren(cl.mindex, level).multipole);
```

The *#pragma omp task* directive from the revision 3 of the standard allows one to design

schemes based on explicit tasks. For that, a single thread inserts the tasks and explicitly performs a *#pragma omp taskwait* call before moving to the next loop such as illustrated in Algorithm 4. Using this primitive still induces many synchronizations. The barriers after each parallel section

**Algorithm 4:** *fj-omp3#task#wait*

```
1 function M2M(tree, kernel, level)
2     #pragma omp parallel
3         #pragma omp single
4             foreach cell cl in tree.cells[level] do
5                 #pragma omp task
6                     kernel.M2M(cl.multipole,
7                                tree.getChildren(cl.mindex, level).multipole);
8             #pragma omp taskwait
```

indeed require the main thread to wait for all the others before proceeding and creating the next section. These global barriers guarantee the coherency of the algorithm by ensuring that when a level/operator is computed, all the required data are ready and no race-conditions are possible. However, they still lead to a fork-join approach. We hence call *fj-omp3#task#wait* (fork-join scheme based on the *#pragma omp task* directive from the revision 3 of the standard) this algorithm in the sequel.

## 4.2 Task-based schemes

The main strength of task-based paradigms is to remove global synchronizations and potentially execute a task as soon as its predecessors (the tasks it depends on) are completed (and, of course, that a resource is available to process it). From this point of view, the introduction of the *#pragma omp task* directive in the version 3.1 of the standard can be viewed as a partial support for task-based scheme that the *depend* clause completed only with the revision 4.0. In the sequel, we will therefore refer to task-based schemes only those either relying on OpenMP 4.0 *task* **and** *depend* constructs or equivalent low-level native runtime directives (StarPU in our case). We name this scheme *tb-omp4#task#dep* for short, whether or not it is implemented with OpenMP directives or native StarPU constructs.

### 4.2.1 Task-based scheme specific granularity control

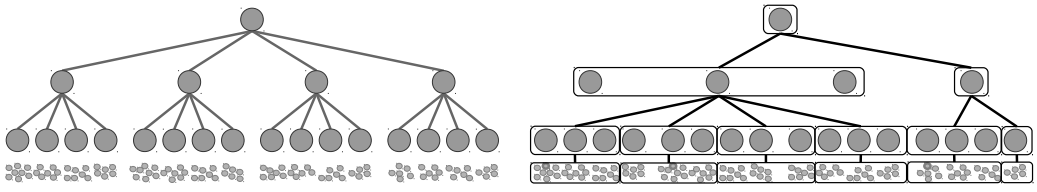


Figure 4: Group tree example for $n_g = 3$.

We showed in [5] that the overhead of task management however highly penalizes task-based FMM approaches (at least when using a fully-featured runtime support such as StarPU) and we proposed to increase the granularity of tasks in order to limit their number. To do so, we introduced a new data structure, the group tree, which is an octree where a number $n_g$ (the *group*

*size*) of consecutive leaves or cells following the Morton index are allocated together and seen as a single element. Therefore, the same cells/particles exist in an octree and a group tree, but the tasks and subsequent dependencies are not on the cells/leaves but on groups of cells/leaves. A simplified representation of a group tree is shown in Figure 4.

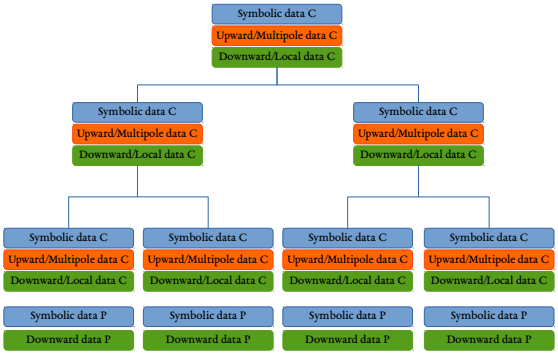



Figure 5: Group tree memory block view, where $C$ and $P$ refer to cells and particles respectively.

Since the scope of the present study aims at discarding complex parallel implementations of the FMM, we furthermore introduce a new design of the group tree from which we can naturally extract extra parallelism. In this new design, the symbolic and numerical data associated with the particles are split. In the case of cells, the numerical data associated with the downward and upward passes are also split. Figure 5 illustrates this new design of the group data structures we have implemented in ScalFMM for the purpose of the present study. We have implemented

**Algorithm 5:** *tb-omp4#task#dep* scheme with OpenMP 4.0 directives

```
1  function FMM(tree, kernel)
2      #pragma omp parallel
3          #pragma omp single
4              // Near-field
5              P2P_taskdep(tree, kernel);
6              // Far-field
7              P2M_taskdep(tree, kernel);
8              for l = tree.height-2 → 2 do
9                  M2M_taskdep(tree, kernel, l);
10             for l = 2 → tree.height-1 do
11                 M2L_taskdep(tree, kernel, l);
12             for l = 2 → tree.height-2 do
13                 L2L_taskdep(tree, kernel, l);
14             // Merge
15             L2P_taskdep(tree, kernel);
16             #pragma omp taskwait
17 function M2M_taskdep(tree, kernel, level)
18     foreach cell cl in tree.cells[level] do
19         #pragma omp task depend(inout:cl.multipole) \\
20                          depend(in:tree.getChildren(cl.mindex, level).multipole)
21         kernel.M2M(cl.multipole, tree.getChildren(cl.mindex, level).multipole);
```

**Algorithm 6:** *tb-omp4#task#dep* scheme with native StarPU directives

```
1  function FMM(tree, kernel)
2      // Near-field
3      P2P_starpu(tree, kernel);
4      // Far-field
5      P2M_starpu(tree, kernel);
6      for l = tree.height-2 → 2 do
7          M2M_starpu(tree, kernel, l);
8      for l = 2 → tree.height-1 do
9          M2L_starpu(tree, kernel, l);
10     for l = 2 → tree.height-2 do
11         L2L_starpu(tree, kernel, l);
12     // Merge
13     L2P_starpu(tree, kernel);
14     starpu_task_wait_for_all()
15 function M2M_starpu(tree, kernel, level)
16     foreach cell cl in tree.cells[level] do
17         starpu_insert_task (READ_WRITE, cl.multipole,
18                             READ, tree.getChildren(cl.mindex, level).multipole,
19                             &kernel.M2M);
```

two versions of this proposed *tb-omp4#task#dep* scheme. Algorithm 5 shows a version written with OpenMP constructs while Algorithm 6 shows the equivalent code written with StarPU directives.

## 4.3 Runtime support

The question we address in this paper is whether OpenMP or OpenMP-like FMM codes can achieve a performance competitive with an optimized code natively written with a modern runtime system (StarPU in our case). For that, we considered some OpenMP codes written in the spirit of historical OpenMP constructs (Section 4.1) as well as new compact task-based algorithms based on modern directives (Section 4.2). We now present the runtime support for those schemes.

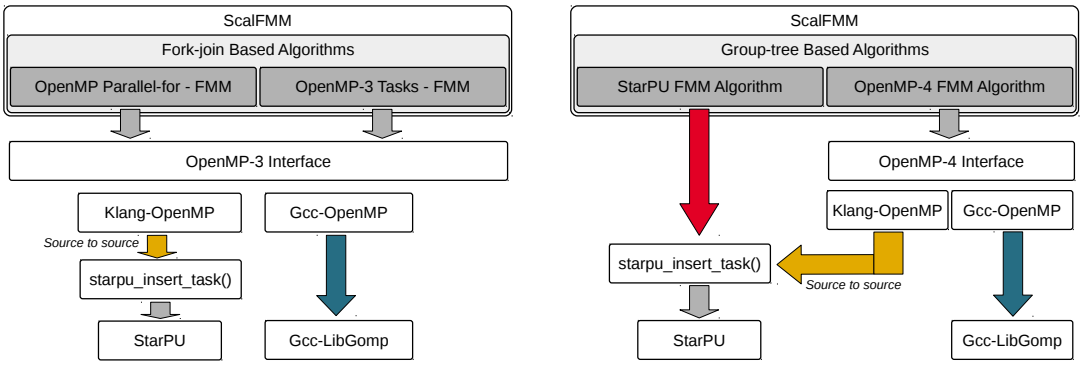
One option for executing those algorithms is to rely on a lightweight dedicated OpenMP runtime system such as the reference LibGOMP library provided with GCC (Gcc/LibGomp). All fork-join schemes and task-based OpenMP schemes discussed in sections 4.1 and 4.2, respectively, can be executed with this support. The paths going through the blue arrows in figures 6a and 6b show the corresponding software stacks. Another option for executing the task-based algorithms proposed in Section 4.2 is to execute their implementations written with native runtime directives (such as Algorithm 6) directly on top of the task-based StarPU runtime, as shown with the path going through the red arrow in Figure 6b.

We furthermore propose to bridge the gap between those two cases by executing OpenMP codes using a native task-based runtime system (StarPU in our case). For that, we rely on the source-to-source Klang/StarPU compiler[1] which translates an OpenMP code into a StarPU code. The most straightforward translation is the one of task-based algorithms. Indeed, we can readily transforms OpenMP tasks (called *explicit* tasks in the OpenMP specification) into StarPU tasks. For instance, Algorithm 5 can be converted by the compiler to be strictly

[1] Klang/StarPU is presented for the first time in the present article, see `http://kstar.gforge.inria.fr/` for details.

equivalent to the native StarPU source code proposed in Algorithm 6. The path going through the yellow arrow in Figure 6b shows the corresponding software stack. However, the OpenMP execution model specification differs slightly with the native StarPU execution model with respect to the *main* thread. In the native StarPU execution model, the *main* thread only executes the sequential part of the application: it inserts tasks and eventually waits for their completion, but it is not bound to any core and is not involved in computing any tasks. Therefore, this thread may freely move over the cores and, though the cost of submitting tasks is often low in regard of their execution cost, it still may visibly alter the execution of the *worker* threads computing light workload tasks, in regard of which the submission time is not negligible. The OpenMP execution model specifies that the *main* thread also acts as a *worker* thread: it takes an active part in executing tasks and is bound to a core. Thus, we extended StarPU such that an OpenMP compliant execution model is enforced when StarPU is used through the Klang/StarPU compiler, with respect to the *main* thread. The execution model of StarPU when used through its native directives has been left unmodified however, which may in some cases account for slightly different execution behaviour between native StarPU programs and Klang/StarPU programs. An example of such a slightly differing behaviour will indeed be observed on the task-based schemes detailed timing study in Sec. 5.4.3.

The fork-join *fj-omp3#task#wait* scheme based on OpenMP 3 independent tasks (Algorithm 4) is also readily implemented by mapping those explicit tasks onto StarPU tasks while furthermore ensuring the `taskwait` synchronizations with starpu_task_wait_for_all() barriers. The language specification also states that fork-join parallel regions are logically expressed as one task (called *implicit* task) for each participating thread. Klang/StarPU takes advantage of this statement to express OpenMP fork-join models as a direct mapping of these implicit tasks on StarPU tasks, following the path through the yellow arrow in Figure 6a. The loop scheduling attribute of an OpenMP parallel for region further specifies how the iterations of the parallel loop are assigned to the participating implicit tasks. The *static* scheduling version, as in Algorithm 3, statically assigns chunks of the iteration range to each participating task. The *dynamic* scheduling version, as in Algorithm 2, instead lets the participating tasks contend at runtime to pick iteration range chunks. Thus, the *static* loop scheduling advantage is to incur less processing overhead, while the advantage of the *dynamic* scheduling is to enable better load balancing if the workload is not uniform across the iteration range.




(a) Fork-join algorithms.

(b) Task-based algorithms.

Figure 6: Software layers.

### 4.4 Enhancing task-based schemes with priority and commutativity

We [5] showed that expressing fine-grain dependencies such as the ones encoded by the DAG in Figure 3b and relying on a group scheme could achieve high performance in general, but could be further accelerated with additional hints. The first required hint, to maintain high performance on a large range of configurations, consists in ensuring a swift progress along the critical path. For that, we attribute priorities to the tasks as defined in Table 1. In the native StarPU case, this leads to slight changes when creating tasks as shown in Algorithm 7 for the M2M operator. Note that we distinguish the priorities between computation within (-inner) and between (-outer) groups (see [5] for more details).

| Operator/Task | P2M | M2M | P2P | M2L-inner* | M2L-outer* |
|---|---|---|---|---|---|
| Priority | 0 | 1 | 2 | $\ell + 1$ | $\ell + 2$ |
| Operator/Task | L2L* | P2P-outer | M2L-inner($h-1$) | M2L-outer($h-1$) | L2P |
| Priority | $\ell + 3$ | $(h-3)*3+2$ | $(h-3)*3+3$ | $(h-3)*3+4$ | $(h-3)*3+5$ |

Table 1: Priorities for the different operators/tasks.

**Algorithm 7:** *tb-omp4#task#dep+p* scheme with native StarPU directives

```
1 function M2M_starpu(tree, kernel, level)
2     foreach cell cl in tree.cells[level] do
3         starpu_insert_task (PRIORITY, M2M_PRIO,
4                             WRITE, cl.multipole,
5                             READ, tree.getChildren(cl.mindex, level).multipole,
6                             &kernel.M2M) ;
```

In Klang/StarPU, we follow the OpenMP specification to support priorities in the task declaration. Algorithm 8 shows the use of *priority* clause so that the original code is only marginally modified. We name *tb-omp4#task#dep+p* the resulting code and we emphasize that

**Algorithm 8:** *tb-omp4#task#dep+p* scheme with OpenMP 4.5 directives

```
1 function M2M_taskdep(tree, kernel, level)
2     foreach cell cl in tree.cells[level] do
3         #pragma omp task priority(M2M_PRIO) depend(inout:cl.multipole) \\
4                          depend(in:tree.getChildren(cl.mindex, level).multipole)
5         kernel.M2M(cl.multipole, tree.getChildren(cl.mindex, level).multipole);
```

it is compatible with the OpenMP 4.5 standard.

Another limit highlighted in [5] is that the order in which the tasks are created in Algorithm 5 is extremely sensitive. Different valid task creation orderings, for this algorithm, lead to different instantiations of the DAG. The fundamental reason is that the `inout` dependence mode specified for data element being updated by a task, indeed not only forces mutual exclusion of tasks with respect to that dependence data, it, furthermore, forces sequential consistency as well. For instance, if a task $T_1$ and a task $T_2$ are submitted in that order to update a piece of data $A$, and if $T_2$ becomes ready to run before $T_1$, $T_2$ will nevertheless *not* be allowed to start before $T_1$ is completed. This results in tasks accessing that piece of data being forcibly executed in the task creation order. For many algorithms, this is the expected behaviour. For FMM Algorithm 5, however, this overconstrains the set of valid task schedules: The Algorithm accumulates — for a given cell — the incoming contributions from the cell's neighbour particles or cells. This

accumulation is inherently commutative, since incoming contributions may update a cell in any order, meaning that multiple execution orders are valid. As a consequence, forcing an arbitrary execution order may unnecessarily delay the execution of ready tasks, wasting parallelism.

To achieve high performance in [5], despite these overconstraining `inout` dependencies, we resorted to implementing complex loop unrolling right in the application code for tuning the tree traversal and subsequently the obtained DAG. This method would however not fit the programming productivity objective of the present study, to achieve high performance, while preserving a concise and elegant application code.

Moreover, while one might think about using a reduction construct, this would not be the best option here, since parallel reductions are implemented using per-thread privatized buffers and a parallel tree-shaped reduction scheme. Parallel reductions are a win when the overall application parallelism is *limited*, that is, when the memory footprint and cost of the extra privatized buffer management is hidden by the extra parallelism offered by the reduction. In the FMM case, the large number of particles and cells being concurrently updated brings a large amount of parallelism by itself. Thus, using a parallel reduction would result in large memory footprint and processing costs from the privatized buffer management, for little parallelism gain and significant performance penalty.

Therefore, we propose to introduce a new data dependence mode, to express commutative update operations. The commutative update mode still enforces mutual exclusion on the piece of data being updated, but relaxes the ordering of the incoming updates. From the application point of view, this involves modifying the task insertion step as show, for instance, with a M2M task on Algorithm 9 (example given for the native StarPU API. Although the OpenMP board

**Algorithm 9:** *tb-omp4#task#dep+c* scheme with native StarPU directives

```
1 function M2M_taskdep(tree, kernel, level)
2     foreach cell cl in tree.cells[level] do
3         starpu_insert_task (COMMUTE, cl.multipole,
4                             READ, tree.getChildren(cl.mindex, level).multipole,
5                             &kernel.M2M);
```

decided to introduce a reduction mode in future revisions, there is (at the time of this writing) no clause expressing the commutative update dependence mode for the time being. We propose to add a *commute* data dependence mode to the *depend* clause, for expressing commutative update operations in OpenMP. Algorithm 10 shows the concise impact on the resulting code. We also

**Algorithm 10:** *tb-omp4#task#dep+c* scheme with OpenMP extensions

```
1 function M2M_taskdep(tree, kernel, level)
2     foreach cell cl in tree.cells[level] do
3         #pragma omp task depend(commute:cl.multipole) \\
4                          depend(in:tree.getChildren(cl.mindex, level).multipole)
5         kernel.M2M(cl.multipole, tree.getChildren(cl.mindex, level).multipole);
```

implemented a support of the proposed *commute* data access mode within Klang/StarPU to assess the resulting *tb-omp4#task#dep+c* (or *tb-omp4#task#dep+cp* when combined with *priority*) code. As a result, the underneath runtime system is not provided with a static DAG as the one from Figure 3b but can instantiate it dynamically depending on the order in which tasks are completed. Figure 7 illustrates the dynamic opportunities that are encoded by the data structure that substitutes that DAG. The resulting compact code can bridge the gap between

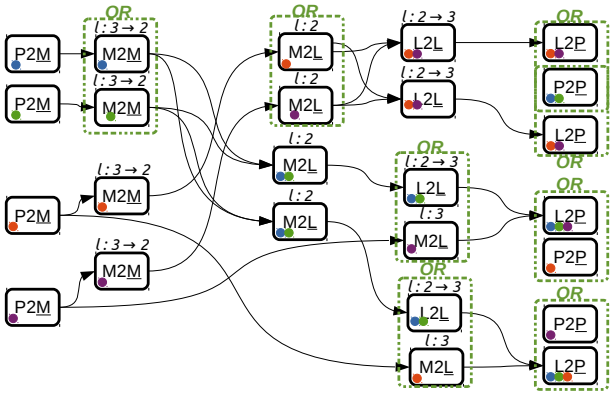



Figure 7: Symbolic meta-DAG for the FMM associated with Figure 3b where tasks with an *OR* block can be executed in any order instantiated dynamically.

OpenMP and native task-based runtime systems, allowing for maximizing both productivity (discussed in this section) and performance (discussed below).

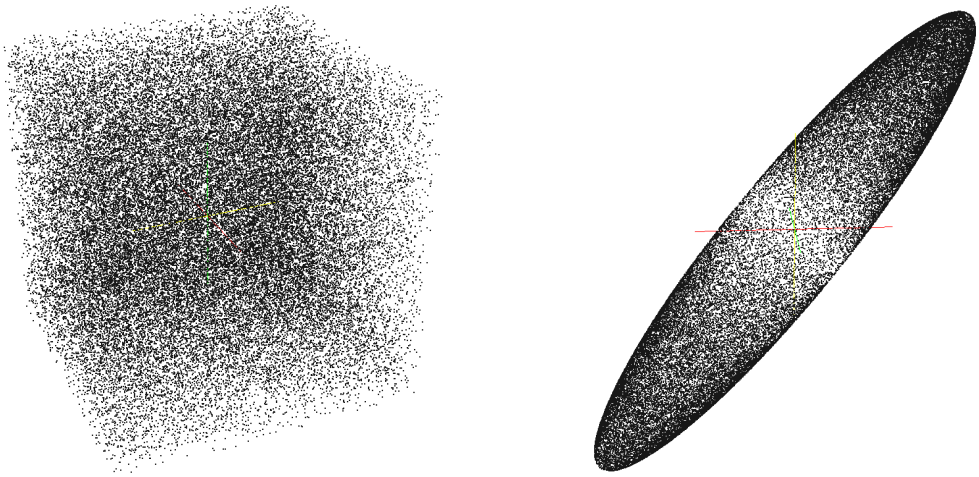
# 5 Experimental study

## 5.1 Experimental setup

We illustrate our discussion with two types of particle distributions assessed on two multicore platforms. The first machine is a 24 cores platform (referred to as *24C* in the sequel) composed of 2 dodeca-core Haswell Intel Xeon E5-2680 processors running at 2.8GHz and with 128GB of shared memory. It can be considered has a common modern multicore processor; we will therefore mainly focus on this platform. The second one is a 96 cores platform (referred to as *96C* in the sequel) composed of 4 × 2 dodeca-core Haswell Intel Xeon E5-2680 running at 2.8GHz and with 132GB of shared memory. This second machine can be viewed as an extreme multicore processor but is interesting for assessing the validity of our claims, especially for the task-based approach which is expected to deliver a higher level of scalability. We use StarPU 1.3 (5/4/2016), Klang/StarPU 1.0 (1/4/2016), ScalFMM 1.4 (5/10/2016), the GNU compiler GCC 4.9, the Intel MKL Blas & FFTW 11.2 on both platforms.

We consider two types of particle distributions. *Cube (volume)* distributions (Figure 8a) are composed of particles uniformly distributed in a unit box, leading to a regular octree and a high and well balanced amount of work for each cell/leave. *Ellipsoid (surface)* distributions (Figure 8b) are composed of particles distributed on the surface of an ellipsoid with a high density at the poles, leading to an irregular octree and highly variable amount of work associated with the nodes of the octree.

We tuned the FMM as follows. For all our simulations, we use the interpolation based FMM on equispaced grid points (UFMM) from [9] on the Laplacian kernel with an order $l = 5$ corresponding to an intermediate accuracy of $10^{-5}$ (see [9] for details). The choice of the height $h$ of the octree balances the amount of work between the near and far fields. We select the height $h$ that minimizes the sequential execution time. In the sequel, we focus on the parallel behavior, but we provide here some sequential execution times using *fj-omp#for-dyn* to solve the FMM for different numbers of particles ($N$) to allow the reader for having some orders of magnitude in mind and better understand the challenge their parallelization represents. The FMM solution for



(a) Cube (volume).

(b) Ellipsoid (surface).

Figure 8: Both types of particle distribution considered in this study.

a cube distribution is obtained in about 57 $s$ and 1100 $s$ in sequential for a number $N$ of particles equal to $N = 10^6$ and $N = 10^8$, respectively. In the ellipsoid case, the solution requires about 6 $s$ and 980 $s$ in sequential for a number $N = 10^6$ and $N = 10^8$ of particles, respectively. The *fj-omp#for-dyn* algorithm is parametrized with a chunk size equal to 10 (see Section 5.3). For task-based approaches, the group size $n_g$ is tuned to minimize the parallel execution time on the *24C* machine (see Section 5.4.1) and we use the same value on the *96C* platform. Given the high number of test combination cases studied, and the substantial cumulated execution time of the whole testing campaign, each measure sample is taken only once. Thus, some isolated measured samples may exhibit experiment bias.

## 5.2 Performance metrics

### 5.2.1 Normalized efficiency

A convenient metric commonly employed to assess the success of a parallelization scheme is the *parallel efficiency* [12] (or *efficiency* for short) noted $e(p)$ in the sequel. Given a reference sequential execution time $t_1$ and a parallel execution time $t_p$ with $p$ processing units, it is computed as follows: $e(p) = \frac{t_1}{p \times t_p}$ . The reference time $t_1$ must correspond to the fastest sequential algorithm [12]. When considering a fully-featured numerical library, there are however many factors involved and it might be cumbersome to guarantee that the sequential reference is optimal. As mentioned above, we rely on a highly optimized kernel [9] and we tune the height $h$ of the octree so as to minimize the sequential execution time. In the task-based approach, the group size $n_g$ may however impact the sequential performance. We tuned $n_g$ so that it minimizes the parallel execution time (see Section 5.4.1). Large group sizes could slightly improve the sequential performance on our smaller test cases but we decided to run the sequential and the parallel algorithm with the same group size to make the analysis clearer. Because our goal is to compare multiple parallelization schemes that can be supported with multiple execution supports, we consistently use the same sequential reference over all the study. However, we distinguish the reference used when comparing fork-join schemes with each other (Section 5.3) from

the one used for comparing task-based schemes with each other (Section 5.4.2). The main difference is that task-based schemes rely on group data structures and benefit from a better locality, which tends to improve both the sequential and parallel overall execution time. We compute the normalization reference $t_1$ using GCC/LIBGOMP on *fj-omp#for-dyn* for Section 5.3, and using GCC/LIBGOMP on *tb-omp4#task#dep* for Section 5.4.2. We refer to the resulting normalized $e(p)$ metrics as the "normalized parallel efficiency," or simply as the *normalized efficiency* when there is no ambiguity.

### 5.2.2 Detailed timings

While the parallel efficiency (or in our case the normalized efficiency defined above) is a very convenient measure for assessing the overall success of a considered parallelization, one may want to analyze in more details the behavior of a method. Indeed, multiple effects can lead to a non optimal parallel efficiency. In a task-based code running on top of a runtime system, the time spent over all processes, or *cumulated time* $\mathcal{T}(p)$, can be cast into time spent in performing actual tasks ($\mathcal{T}^{task}(p)$), in runtime management ($\mathcal{T}^{runtime}(p)$) or scheduling ($\mathcal{T}^{scheduling}(p)$) operations, and idle time ($\mathcal{T}^{idle}(p)$). The overall cumulated time $\mathcal{T}(p)$ is homogeneous to *processing unit* $\times$ *second* and satisfies

$$\mathcal{T}(p) := t_p \times p \;=\; \mathcal{T}^{task}(p) + \mathcal{T}^{runtime}(p) + \mathcal{T}^{scheduling}(p) + \mathcal{T}^{idle}(p).$$

Due to potential contention (caches, buses, ...), the actual computation may be slowed down, leading to a higher time spent in tasks ($\mathcal{T}^{task}(p)/\mathcal{T}^{task}(1) > 1$). Furthermore, the runtime system must consume time in order to ensure the overall progress (task insertion, data consistency, ...). In our case, we furthermore distinguish the time spent within the runtime from the time spent in scheduling. We want to assess whether those costs are reasonable with respect to the ideal time spent in tasks ($\mathcal{T}^{runtime}(p)/\mathcal{T}^{task}(1) \ll 1$ and $\mathcal{T}^{scheduling}(p)/\mathcal{T}^{task}(1) \ll 1$). Finally, idle time results from a lack of concurrency possibly combined with suboptimal scheduling decisions. We again assess this effect with respect to the ideal time spent in tasks ($\mathcal{T}^{idle}(p)/\mathcal{T}^{task}(1)$). Consequently, STARPU has been instrumented to separately record the time spent in application tasks, scheduler algorithm, runtime management and in the idle loop, respectively so as to report such detailed timings for both the KLANG/STARPU and the STARPU supports.

## 5.3 Performance of fork-join schemes

This section reports performance results of the fork-join schemes discussed in Section 4.1.

### 5.3.1 Normalized efficiency

Figure 9 and Figure 10 present the parallel efficiencies of the *fork-join* schemes discussed in Section 4.1 normalized by the GCC/LIBGOMP *fj-omp#for-dyn* sequential reference on the *24C* platform for the cube and ellipsoid distributions, respectively. We assess all three *fj-omp#for-dyn*, *fj-omp#for-sta* and *fj-omp3#task#wait* fork-join parallelization schemes proposed in algorithms 2, 3 and 4. Both the GCC/LIBGOMP (blue) and KLANG/STARPU (yellow) source-to-source OPENMP compiler / runtime system frameworks are considered (consistently with the color code of the software stack proposed in Figure 6a). The main observation is that the KLANG/STARPU support for the *#pragma omp parallel for* directive is competitive against the lightweight GCC/LIBGOMP support for both the *fj-omp#for-dyn* and *fj-omp#for-sta* schemes, as long as the number of particles is not extremely low ($N = 10^6$).

The independent tasks support of KLANG/STARPU shows more overhead than GCC/LIBGOMP due to the heavier weight of the underlying STARPU task covers compared to the extremely

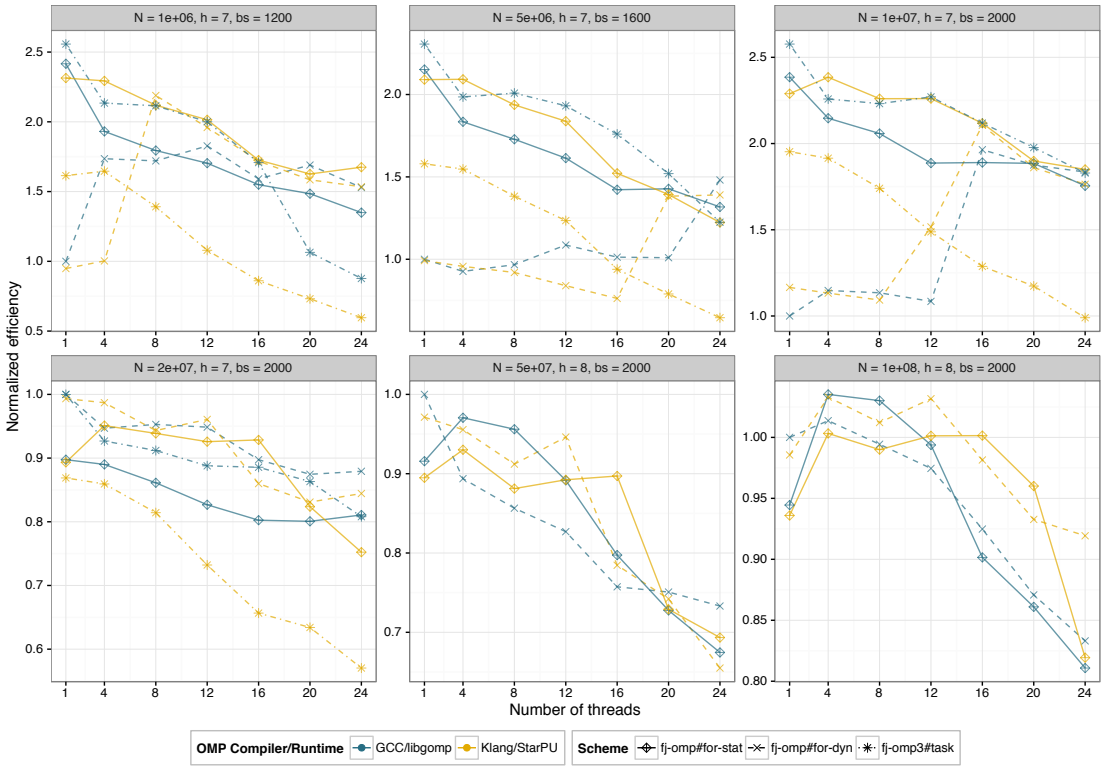



Figure 9: Parallel efficiency normalized by the GCC/LIBGOMP *fj-omp#for-dyn* sequential reference for cube (volume) distributions on the *24C* machine. The different plots do not use the same scale for the *Normalized efficiency* axis. For the largest cases (row 2, col 2–3), the *fj-omp3#task#wait* scheme does not run to completion, due to the overwhelming amount of tasks being created.

lightweight GCC/LIBGOMP ones. This extra cost especially appears on the *fj-omp3#task#wait* scheme, but is also visible on the sequential data point for most schemes. However, this overhead is compensated by a better scalability and smoother behavior of KLANG/STARPU task support. These figures also show the interest of the *fj-omp#for-sta* balanced scheme on such small cases, both for GCC/LIBGOMP and KLANG/STARPU. On the contrary, the *fj-omp#for-dyn* scheme is inefficient on these small cases (top/left plots on the cube) since the participating threads heavily contend on work-sharing the small iteration ranges of the parallel for loops.

A large number of particles (bottom plot row on the cube distribution), or a more complex structure prone to load imbalance (all plots on the ellipse distribution), makes the *fj-omp#for-sta* scheme redundant, changing its benefits observed for small cube cases into a penalty here. Conversely the *fj-omp#for-dyn* scheme generally performs better: the larger amount of particles reduces the contention on parallel for iteration ranges, while the dynamic loop scheduling offers better load balancing by design. The *fj-omp3#task#wait* scheme shows mediocre to bad results for GCC/LIBGOMP and worse results for KLANG/STARPU. A large number of independent tasks generates runtime processing overhead without having the opportunity to offer some benefit in return. The phenomenon is further emphasized by increasing the height of the tree, thus, increasing the number of tasks by creating lower levels with much more cells. Moreover, the

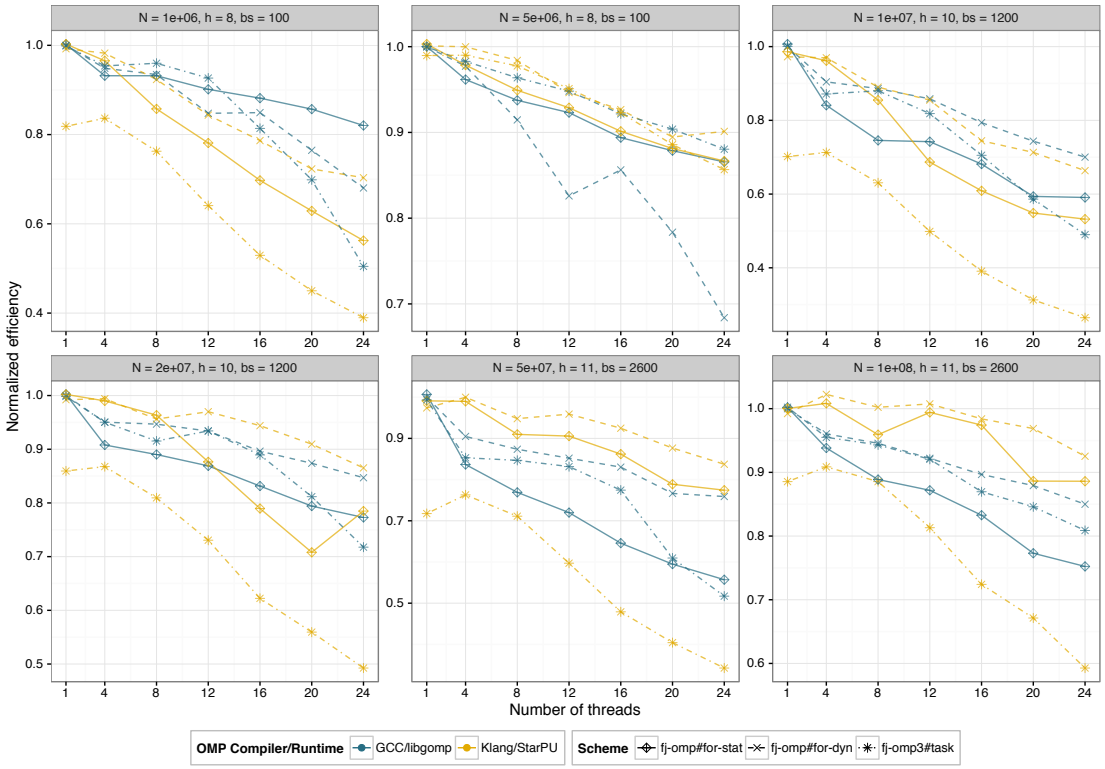



Figure 10: Parallel efficiency normalized by the GCC/LIBGOMP *fj-omp#for-dyn* sequential reference for ellipsoid (surface) distributions on the *24C* machine. The different plots do not use the same scale for the *Normalized efficiency* axis.

*fj-omp3#task#wait* schemes using STARPU and KLANG/STARPU supports fail to compute the largest simulations ($N = 5 \cdot 10^7$ and $N = 10^8$) only on the cube, because the number of generated tasks it too important and lead to out-of-memory issues. Indeed, for $N = 5 \cdot 10^7$ and $h = 8$ there are more than 5M tasks just for the $M2L$ at leaf level.

### 5.3.2 Detailed timings

Figure 11 shows the resulting detailed timings (see Section 5.2.2) for running all fork-join schemes with the KLANG/STARPU support. We see that the *fj-omp#for-dyn* scheme incurs small runtime overhead and idle times. However, the time to compute the tasks increases with the number of threads, since all the threads have to dynamically compete for parallel loops work-sharing. On the contrary, the work-sharing division is done statically with the *fj-omp#for-sta* scheme.

The *fj-omp#for-sta* scheme shows an increasing idle time, though, which is correlated to the $\mathcal{T}^{task}(p)$ task self execution time improvement, as the number of threads increases. This acceleration is mainly due to the resulting increased data locality and reuse, because the work is split into sub-trees where most of parents/children and neighbors are included. Yet, this acceleration in the computation of the tasks also makes the static iteration space split effectively more unbalanced in terms of execution time, which increases the idle time.

Finally, the *fj-omp3#task#wait* scheme shows significant runtime overhead and dramatic in-

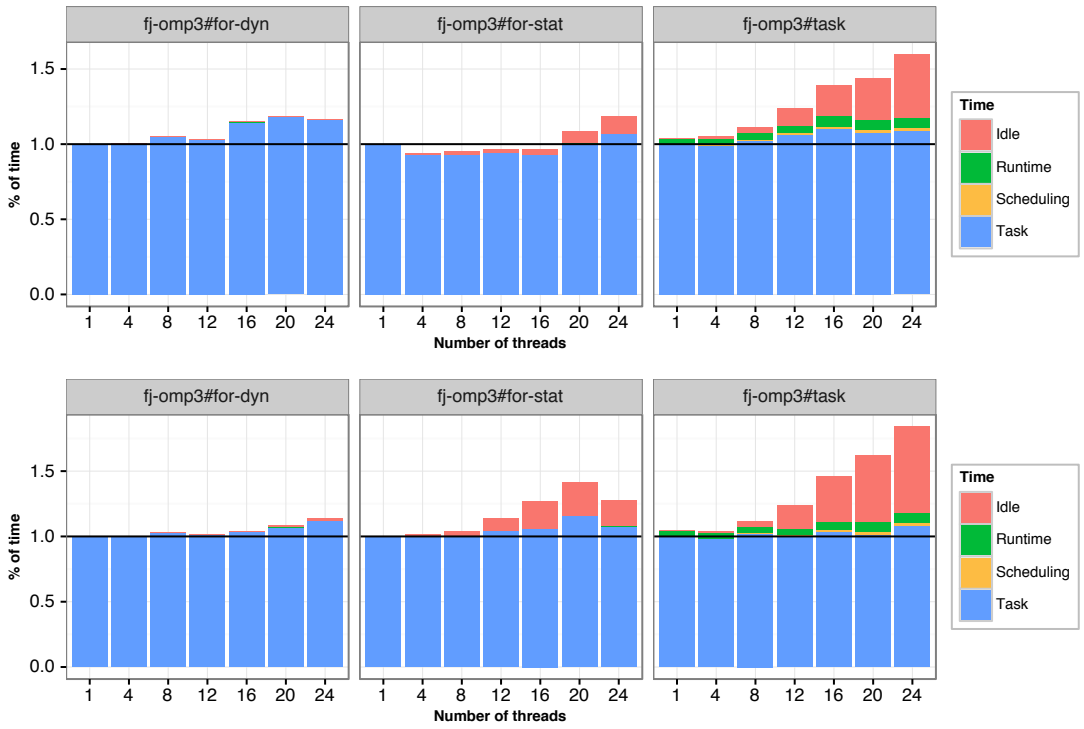



Figure 11: Detailed *fork-join* timings for the cube (volume) distribution, above ($N = 2 \cdot 10^7$, $h = 7$), and the ellipsoid (surface), below ($N = 2 \cdot 10^7$, $h = 10$), with KLANG/STARPU on platform *24C*.

creasing $\mathcal{T}^{task}(p)$ due to the small granularity. We remind that we create one task per operation, which is equivalent to a dynamic parallel for with a chunk size of one. Meanwhile, it confirms the intuition that it is more costly in terms of runtime overhead to manage additional tasks rather than sharing work based on an index increment.

## 5.4 Performance of task-based schemes

This section reports performance results of the task-based schemes proposed in sections 4.2 and 4.4.

### 5.4.1 Granularity

We introduced the group-tree and the associated granularity parameter $n_g$ in Section 4.2. Plots on figure 12 show the measured impact of $n_g$ on the performance of parallel executions relying on GCC/LIBGOMP (blue), KLANG/STARPU (yellow) and STARPU (red), consistently to the software stack proposed in Figure 6b (and according to its color code). As expected when the granularity parameter is too small, all three scheme/support combinations tested perform poorly because the task computational weight is lower and the total amount of tasks generated is higher. Each runtime system introduces per-task managing costs such as allocating data structures, queuing the task, resolving its dependencies and scheduling it; thus, when the task granularity is small, this per-task overhead becomes significant in regard of the task execution time. Moreover,

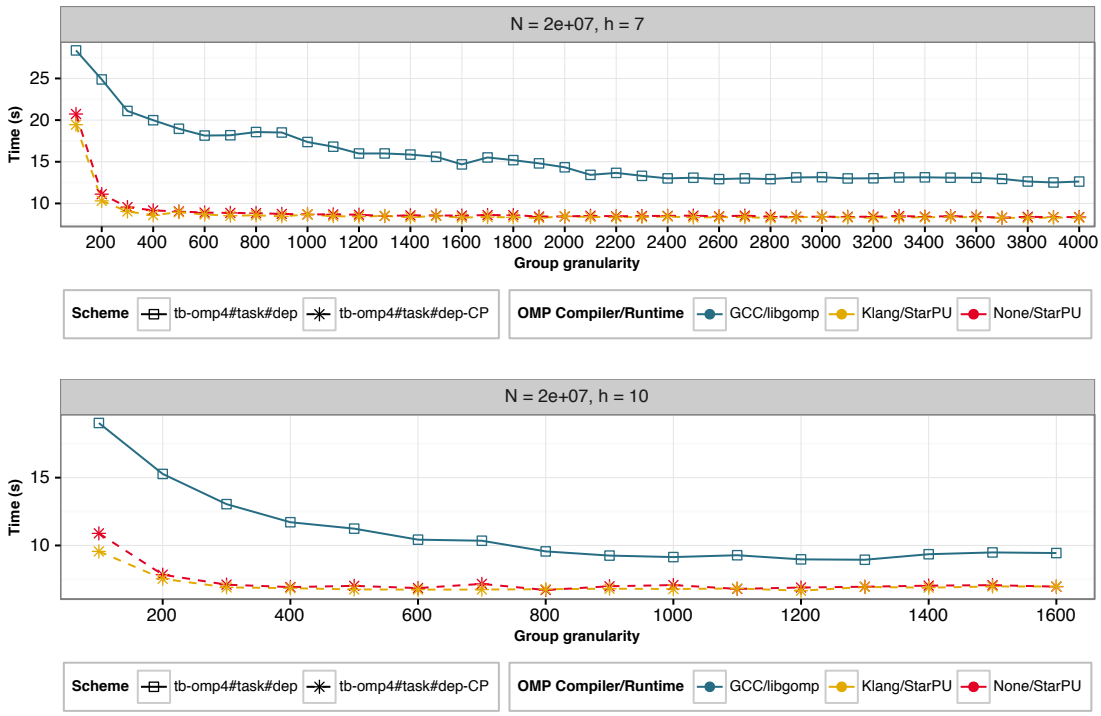



Figure 12: Execution time for the cube (volume) distribution, above, and the ellipsoid (surface), below, for different values of the granularity parameter $n_g$ on platform *24C* using 24 threads.

the larger number of tasks means that this per-task overhead adds-up to a higher penalty on the application execution time. On the other hand, increasing the granularity improves the execution time up to a given limit related to the test case sensitiveness to load imbalance: This limit can therefore especially be observed, for GCC/LIBGOMP, on the ellipsoid distribution plot around $n_g = 1400$, while for the cube distribution, the execution time still remains at the plateau level at a much larger $ng = 4000$. STARPU appears less sensitive to load imbalance at high $n_g$ values for both particle distributions. For each considered particle distribution, we select the granularity that leads to the lowest execution time over all scheme/support combinations on the maximum number of cores available. All executions related to that particle distribution are then applied with this granularity.

### 5.4.2 Normalized efficiency

Figure 13 and Figure 14 present parallel efficiencies of the *tb-omp4#task#dep* scheme using GCC/LIBGOMP (blue), KLANG/STARPU (yellow) and STARPU (red) supports, normalized by the GCC/LIBGOMP *tb-omp4#task#dep* sequential reference on the *24C* platform, for the cube and ellipse distributions, respectively. GCC/LIBGOMP, KLANG/STARPU and STARPU perform very similarly on the *tb-omp4#task#dep* scheme. Again, for a single thread, GCC/LIBGOMP is faster than the other supports, showing that the GCC/LIBGOMP tasks are lighter. The figure also demonstrates the interest of the commutative dependencies extension (see Algorithm 10) on the *tb-omp4#task#dep+c* scheme using either KLANG/STARPU or STARPU. Since the test is performed on a uniform grid distribution of particles, the potential incoming contributions of

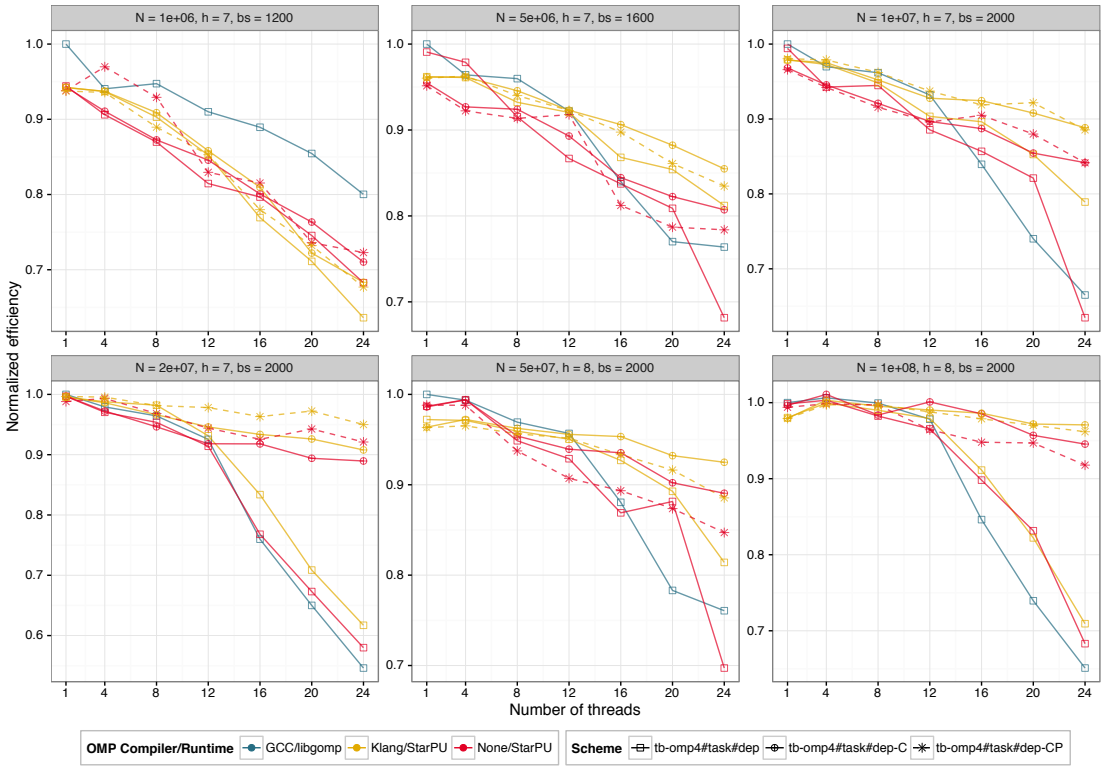



Figure 13: Normalized efficiency for cube (volume) on platform *24C* from 1 to 24 threads. The different plots do not use the same scale for the *Normalized efficiency* axis.

a particle or group are numerous. As the number of threads increases, the ability to integrate these contributions on the fly, in no particular order, leads to a high benefit for the commutative dependencies extension. The *tb-omp4#task#dep* scheme is penalized by forcing the integration of incoming contributions in the sequential data dependence order, regardless of their availability.

If the particle distribution is less uniform, the advantage of the commutative dependencies may however eventually get canceled by the processing overhead it incurs. For instance, when using the ellipsoid distribution and a deep tree, collecting contributions for lots of small, highly-clustered tasks constitutes most of the work. Thus a task does not have to wait a lot for contributions from other tasks to be ready, even when forcing the seemingly overkill sequential data dependence order. This can be seen on Figure 14, running the same comparison on the ellipse distribution.

We further evaluated the benefit of prioritizing tasks, using the *tb-omp4#task#dep+cp* scheme, to favor tasks from the critical path (see Algorithm 8). As shown on both figures 13 and 14, the extra benefit of priorities is not as convincing as the benefit of commutative dependencies was on the uniform particle distribution. Nevertheless, the fact that both the commutative dependencies and the task priorities are supported by KLANG/STARPU makes it straightforward to explore their effectiveness. This is especially valuable in the case of an application such as SCALFMM whose execution behavior highly depends on the dataset characteristics.

Figures 15 and 16 show results obtained on a large shared-memory 96-core machine. For the uniform grid particle distribution (Figure 15), the *tb-omp4#task#dep+c* scheme's commutative

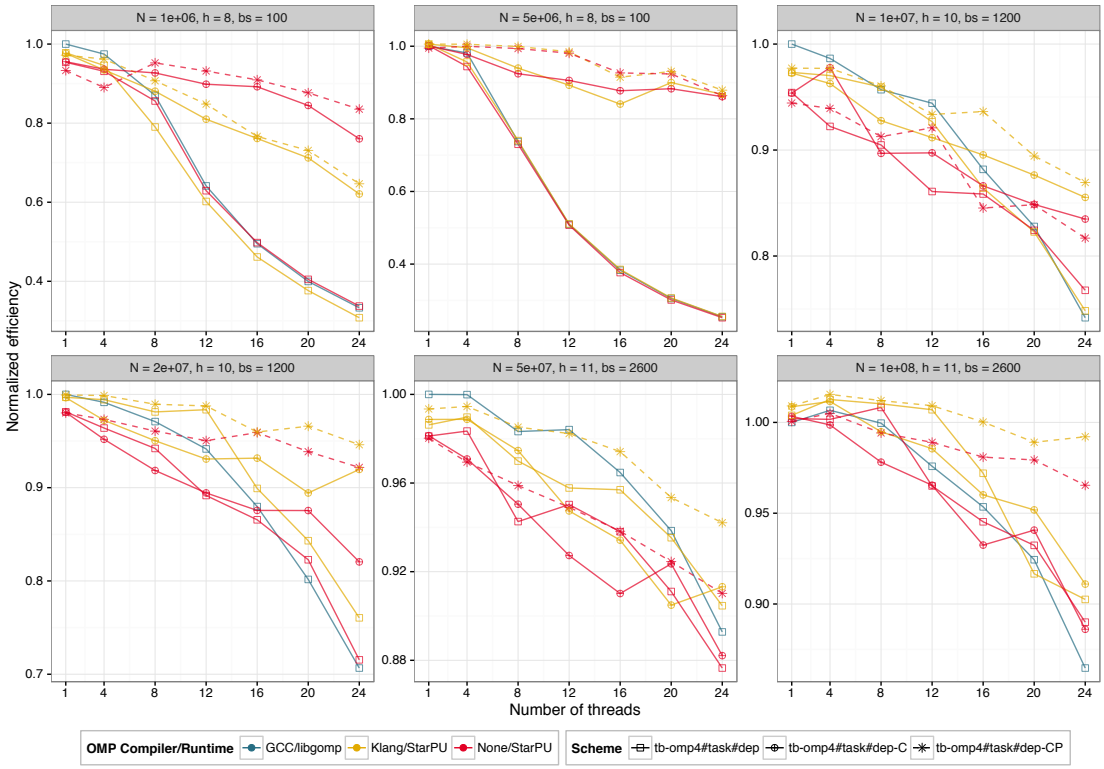



Figure 14: Normalized efficiency for ellipsoid (surface) on platform *24C* from 1 to 24 threads. The different plots do not use the same scale for the *Normalized efficiency* axis.

dependencies shows a much better scalability than *tb-omp4#task#dep*. The very large number of threads stresses the necessity to integrate incoming contributions as soon as possible, which commutative dependencies enable. For the ellipsoid particle distribution, the large number of threads also calls for using commutative dependencies. Moreover, the priorities of scheme *tb-omp4#task#dep+cp* improves the efficiency slightly over *tb-omp4#task#dep+c*. Indeed, while it is important to allow integrating contributions in no particular order, the deep tree structure of this test case however requires processing tasks in a sensible, critical path-aware order for maximizing parallelism.

### 5.4.3 Detailed timings

From KLANG/STARPU detailed timing results in Figure 17, it appears that for all configurations, the overhead of the parallelization compared to the $\mathcal{T}^{task}(p)$ task self execution time is negligible, overall. The idle time showing up beyond 12 threads on the baseline *tb-omp4#task#dep* cases is mainly due to the load imbalance resulting from the overconstrained sequential consistency enforced on concurrent particle/cell updates discussed in Sec. 4.4. In contrast, the effectiveness of the commutative dependencies extensions on reducing idleness is clearly emphasized, especially when used in cooperation with task priorities.

The STARPU detailed timing results in figure 18 may appear surprising at first sight, since the idle time overhead slightly differs from the one obtained with KLANG/STARPU, without

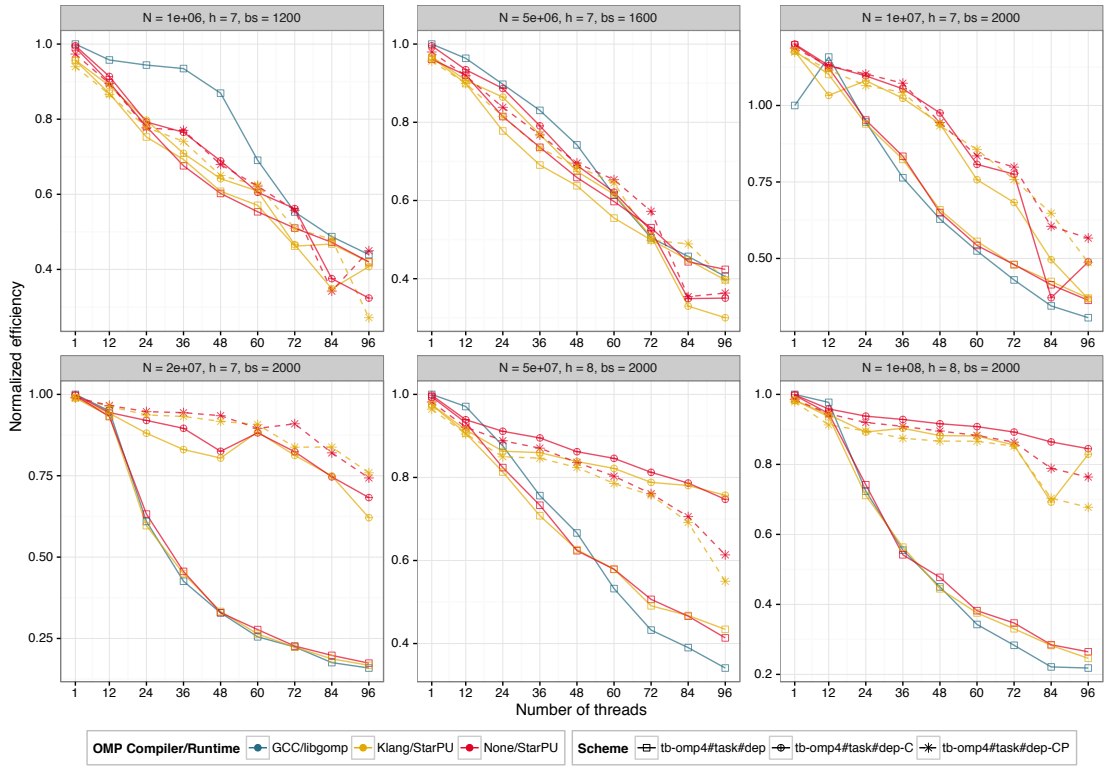



Figure 15: Normalized efficiency for cube (volume) on *96C* from 1 to 96 threads. The different plots do not use the same scale for the *Normalized efficiency* axis.

a seemingly logical scheme. This comes from a behavioural difference of the native execution model of StarPU and the OpenMP compliant execution model provided through Klang/-StarPU with respect to the *main* application thread, as discussed in Section 4.3. Both Klang/-StarPU and StarPU supports' detailed results also emphasize the benefit of exploiting the commutativity (*tb-omp4#task#dep+c* scheme) for reducing the idle time, even further enhanced when combined priorities (*tb-omp4#task#dep+cp* scheme) to the point where it almost becomes negligible.

# 6 Conclusion

In this paper, we presented a study of the ScalFMM library port on OpenMP using first a *legacy fork-join* parallelizing approach and then a OpenMP *4* era *dependent tasks* parallelizing approach. Our aim was to explore the benefits and potential performance trade-offs of using an abstract programming layer for improved application programming productivity. We used the GNU Gcc/LibGomp and the Klang/StarPU compiler and runtime couples, as well as the StarPU runtime used natively through its dedicated API, to conduct an extensive campaign of performance evaluations. We explored using two vastly different numerical setting inputs, both on a 24-core platform and a large 96-core platform. The results first showed the interest of the OpenMP 4 dependent tasks over the legacy fork-join model, especially in the case of the

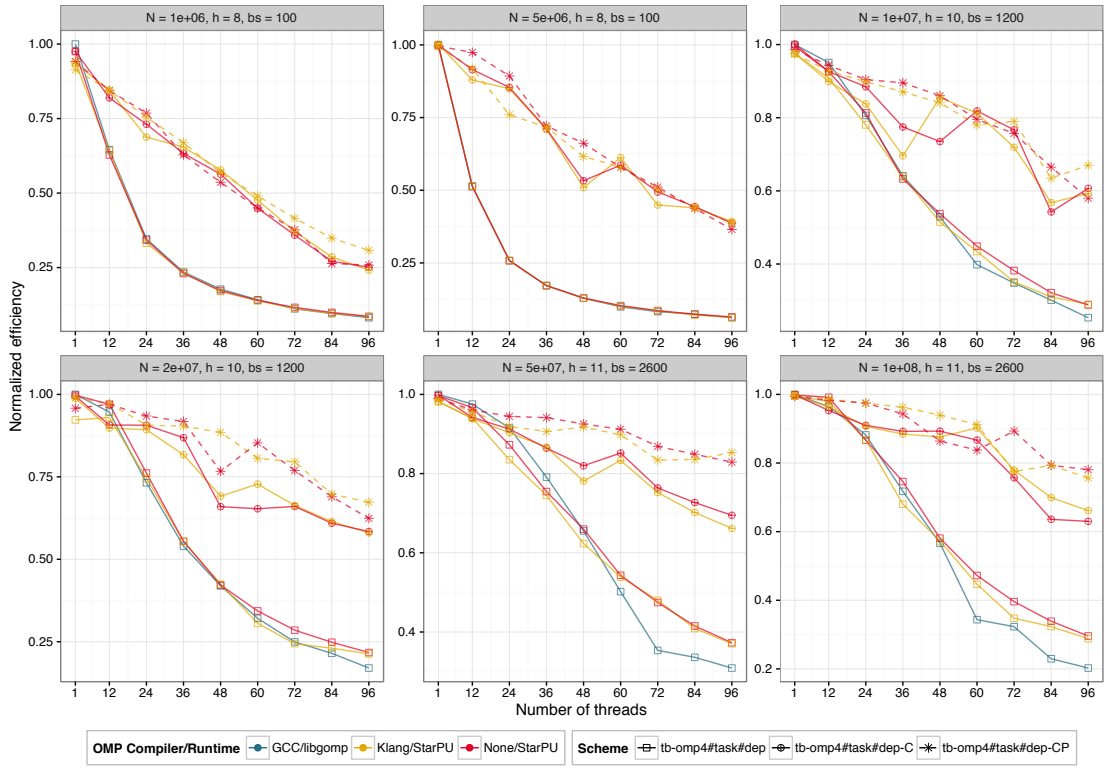



Figure 16: Normalized efficiency for ellipsoid (surface) on *96C* from 1 to 96 threads. The different plots do not use the same scale for the *Normalized efficiency* axis.

ellipsoid particle distribution. In contrast to the cube distribution — where parallelism is readily abundant, in a largely unstructured form, thus a somewhat "low hanging fruit" — the ellipsoid distribution indeed stresses the ability of the underlying runtime system to harvest and exploit complex parallelism. In particular, it demonstrated the high benefit of pinpointed OpenMP task extensions to access additional runtime system features, such as the support for commutative task dependencies, or to a lesser extent the ability to specify task priorities, as made possible by the Klang/StarPU compiler. On the other hand, GCC+LibGomp proved to be very much competitive for datasets with very small tasks. By providing the abstraction of its programming layer, OpenMP makes it possible to get the best of both worlds. In the future, we intend to explore the coupling of OpenMP with task-based runtime systems further, in particular focusing on the support for heterogeneous CPU+accelerator platforms that could be illustrated with a task-based FMM for heterogeneous machines [6].

Note that the support for priorities, compliant with the version 4.5 of the OpenMP standard has been incorporated into GCC 6.1 when finalizing our (long started) experimental campaign. We expect that GCC+LibGomp will benefit similarly as Klang/StarPU of this hint. Indeed, we showed that both suites are extremely robust and comparable concerning performance without priorities, and we expect that it shall be the same with this additional hint. All in all, indeed, the goal of the paper is not to provide a full comparison of the GCC+LibGomp versus Klang/-StarPU, but rather to show that, because they are both competitive suites, their use is credible for assessing the discussed programming models. As a consequence, we expect that the provided

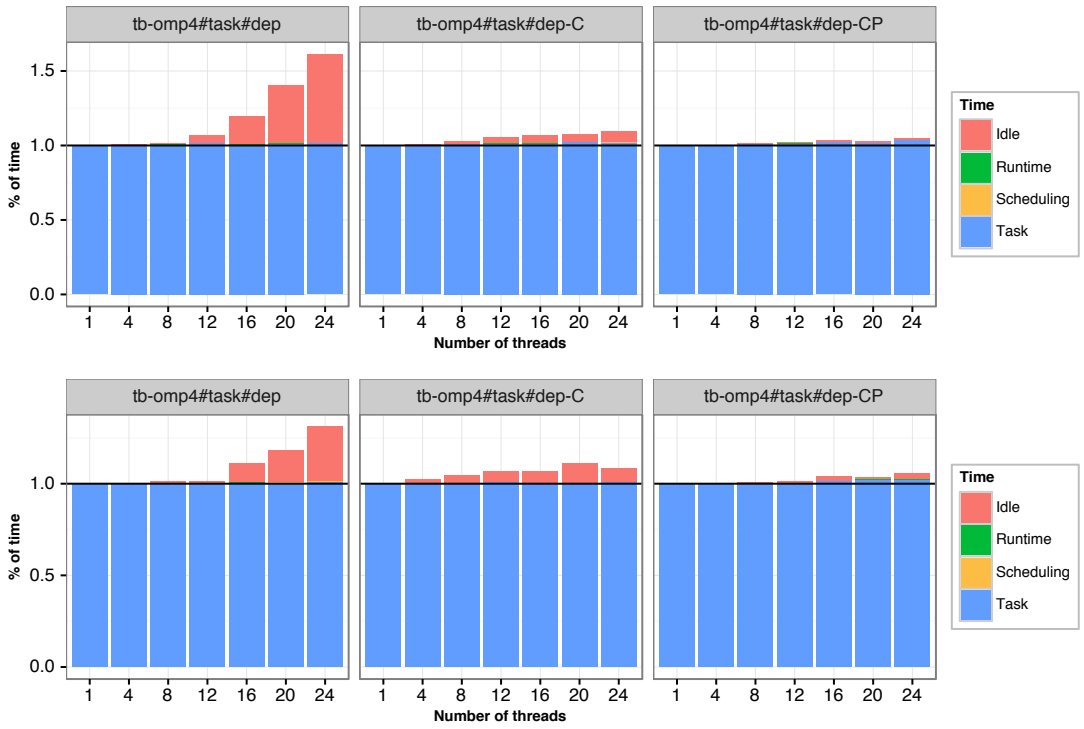



Figure 17: Detailed *task-based* timings for the cube (volume) distribution, above ($N = 2 \cdot 10^7$, $h = 7$), and the ellipsoid (surface), below ($N = 2 \cdot 10^7$, $h = 10$), with KLANG/STARPU on platform *24C*.

experimental results will provide a snapshot of the progress made by the OPENMP standard in bridging the gap over fully-feature runtime systems used natively. OPENMP now allows to exploit modern multicore architectures as efficiently, while preserving both runtime independence and ease of programming. It is succeeding in converging towards a well-balanced standard for writing HPC scientific libraries. On that account, we hope this work will motivate the developers of task-based scientific libraries ([19], for instance, in the case of FMM) to reconsider the potential of OPENMP in achieving high-performance while relying on task-based programming. We also believe that the present analysis might be a valuable feed-back to the OPENMP developers and community, and supports the motivation of integrating the proposed extensions.

Nonetheless, it has recently been shown [13] that deeper static preliminary analysis could enhance the performance of FMM. Providing all the information of such a thorough static pre-processing through OPENMP directives still remains a challenge, that certainly needs to be co-addressed in future work by the OPENMP community and the developers of scientific, high-performance libraries.

# Acknowledgment

Experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux and

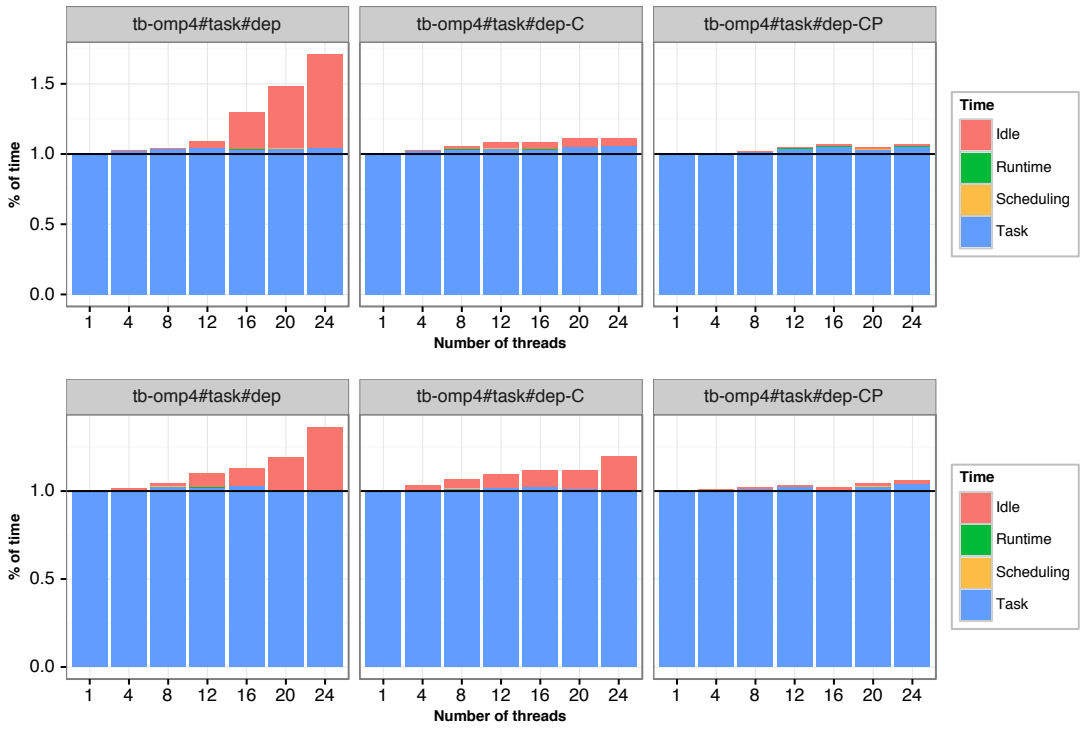



Figure 18: Detailed *task-based* timings for the cube (volume) distribution, above ($N = 2 \cdot 10^7$, $h = 7$), and the ellipsoid (surface), below ($N = 2 \cdot 10^7$, $h = 10$), with StarPU on platform *24C*.

CNRS (and ANR in accordance to the programme d'investissements d'Avenir).

The Klang/StarPU compiler has been developed with the contribution of the ADT k'Star development action from Inria.

# A Appendix

## A.1 Performance of fork-join schemes

### A.1.1 Normalized efficiencies

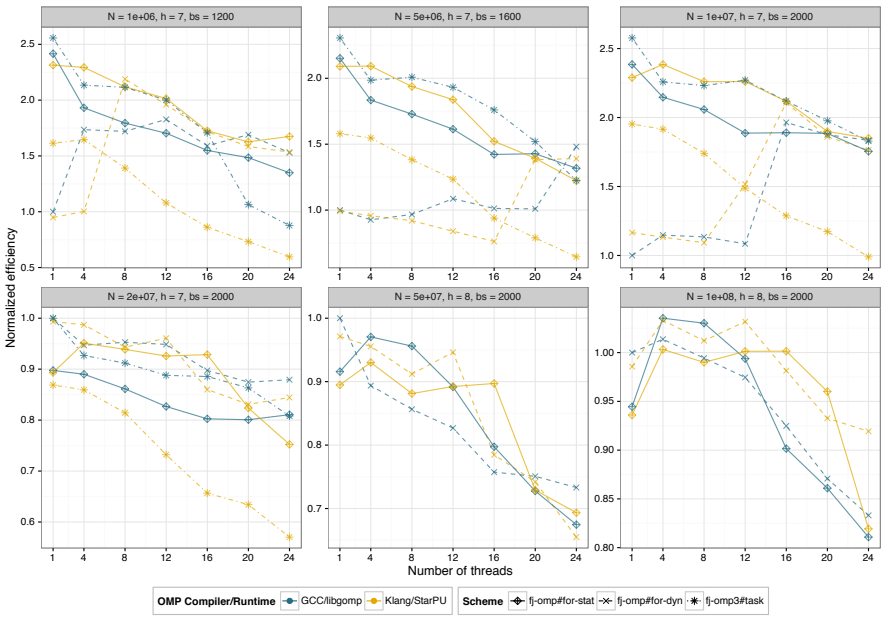



Figure 19: Normalized efficiencies for cube (volume)

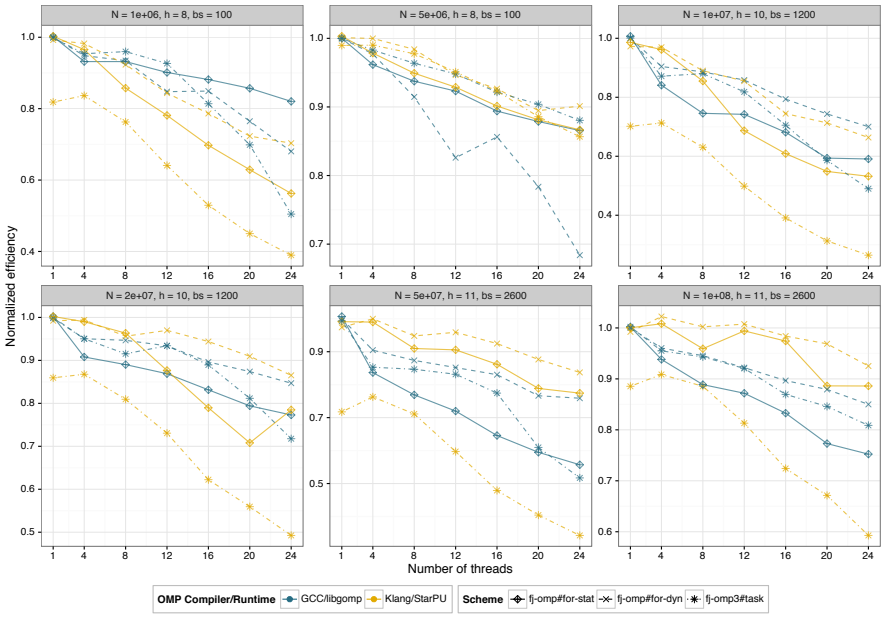



Figure 20: Normalized efficiencies for ellipsoid (surface)

### A.1.2 Speedup

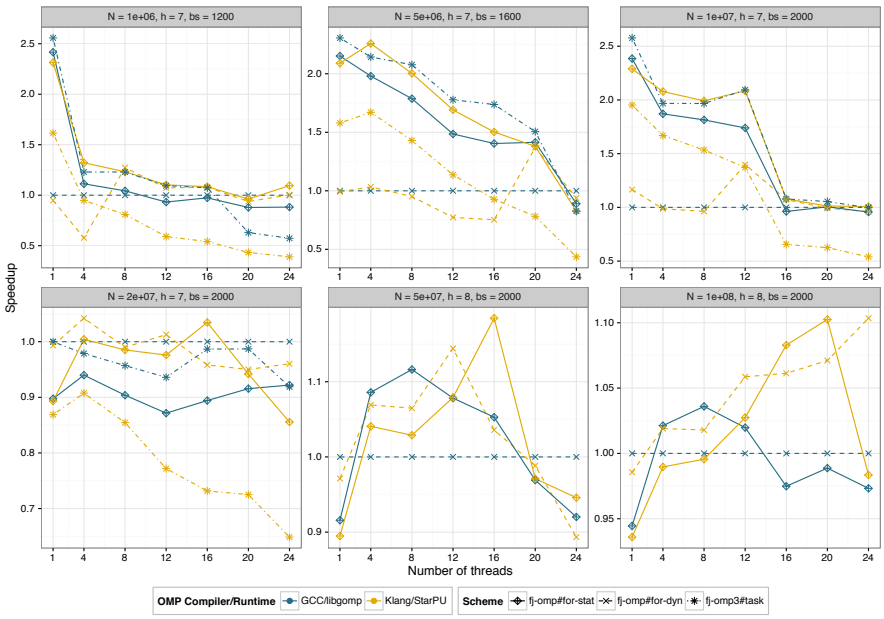



Figure 21: Speedup for cube (volume)

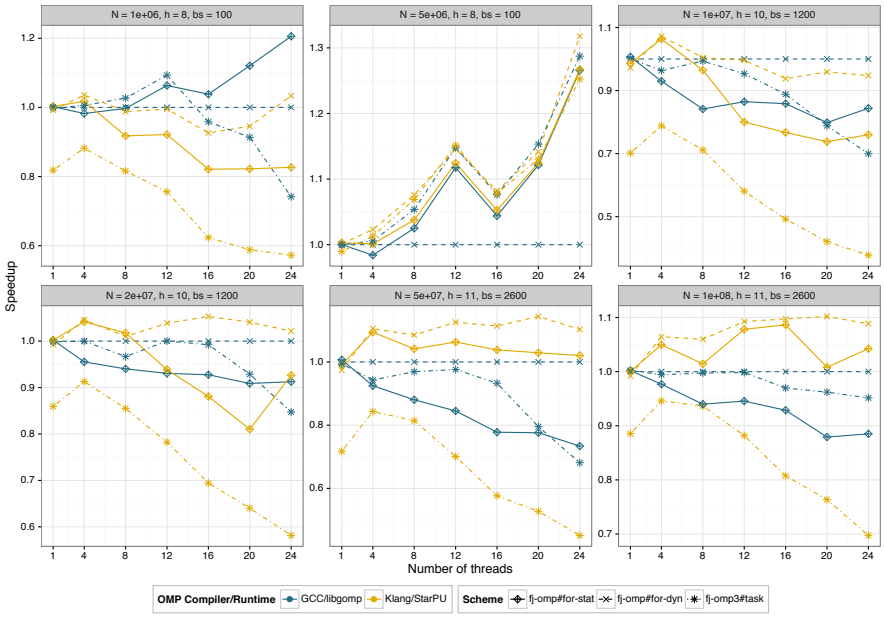



Figure 22: Speedup for ellipsoid (surface)

### A.1.3 Parallel efficiency

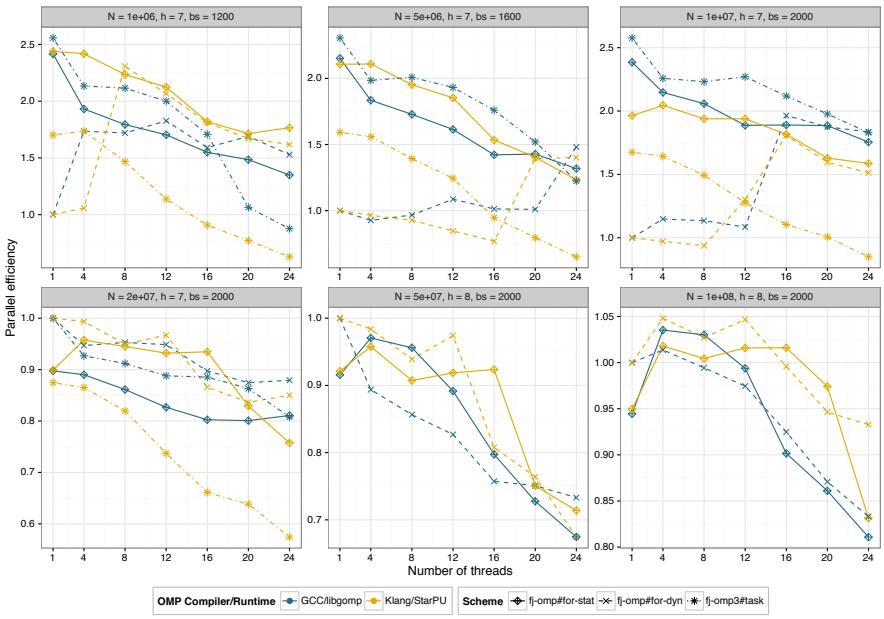



Figure 23: Parallel efficiency for cube (volume)

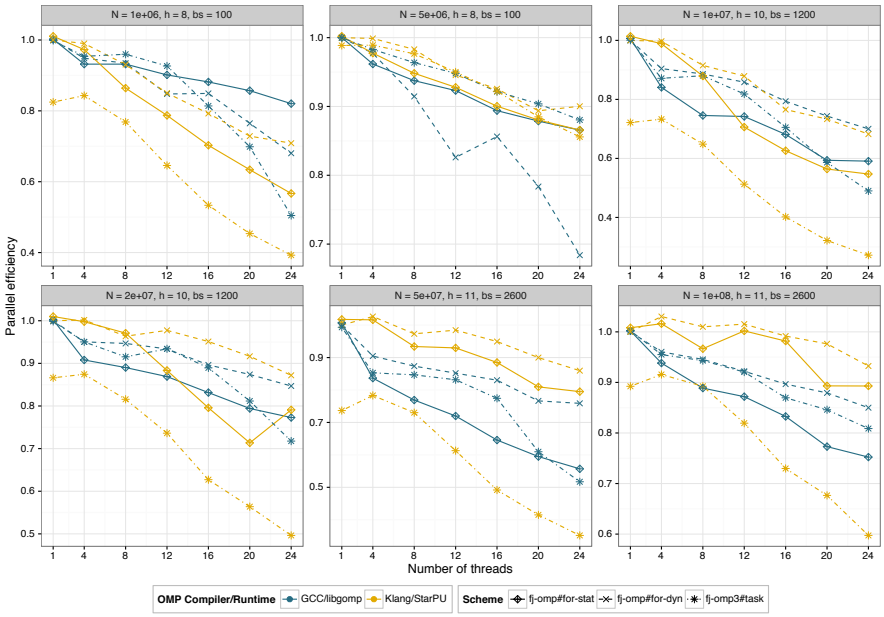



Figure 24: Parallel efficiency for ellipsoid (surface)

### A.1.4 Timings

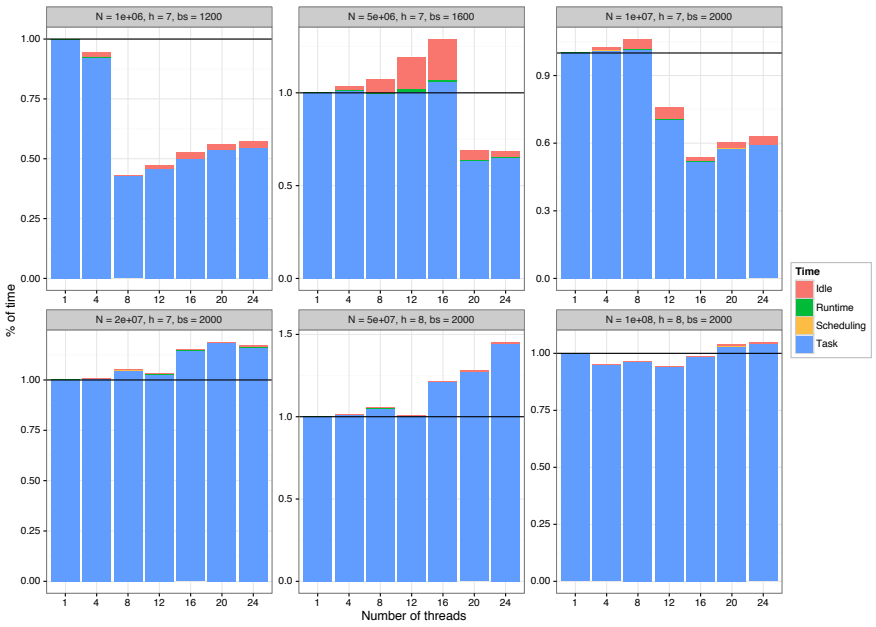



Figure 25: timings for cube (volume) with Klang-for-dyn

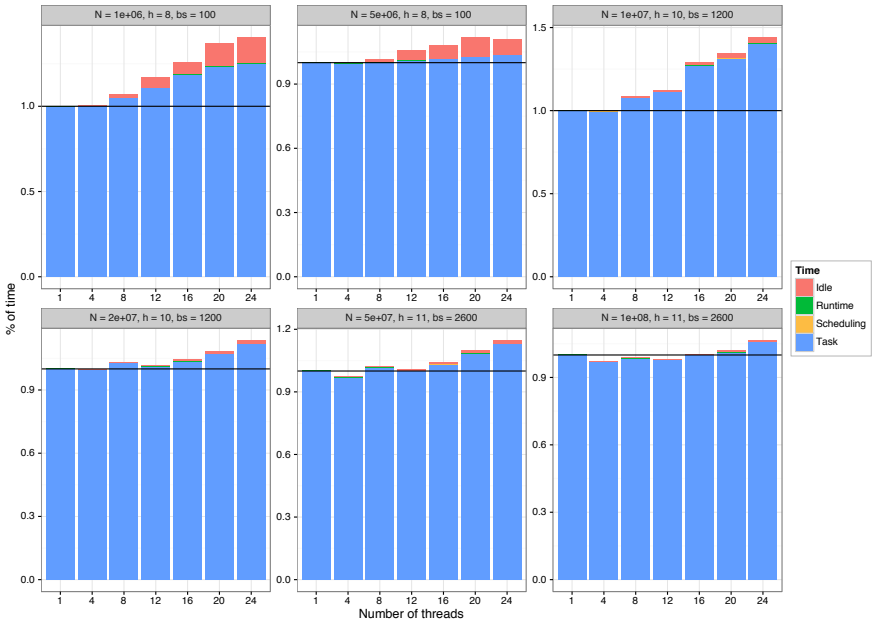



Figure 26: timings for ellipsoid (surface) with Klang-for-dyn

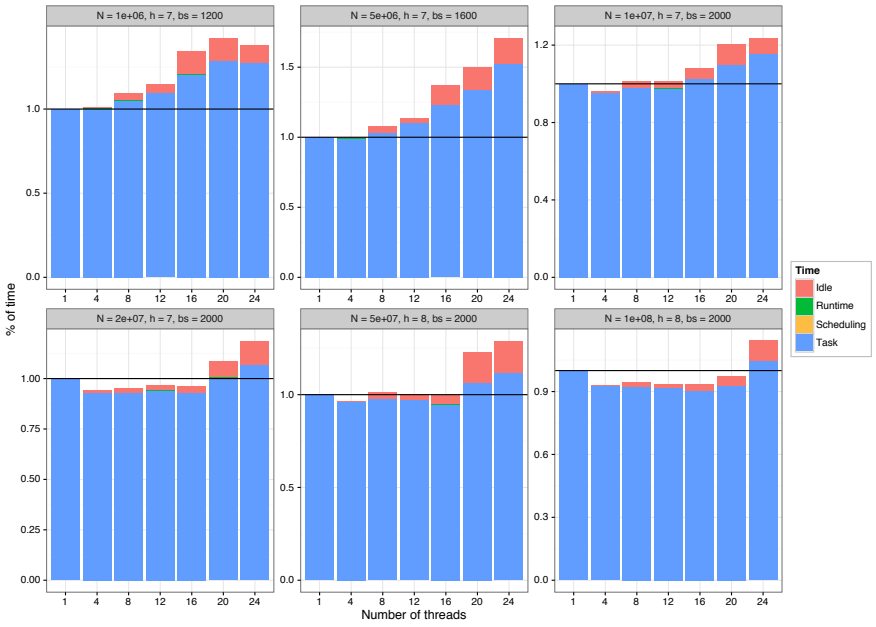



Figure 27: timings for cube (volume) with Klang-for-stat

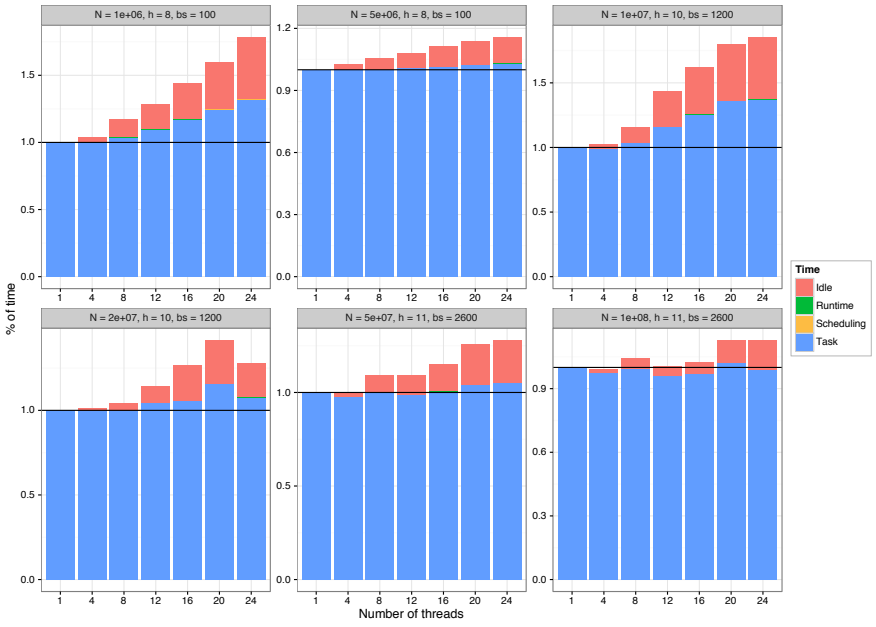



Figure 28: timings for ellipsoid (surface) with Klang-for-stat

### A.1.5 Efficiencies

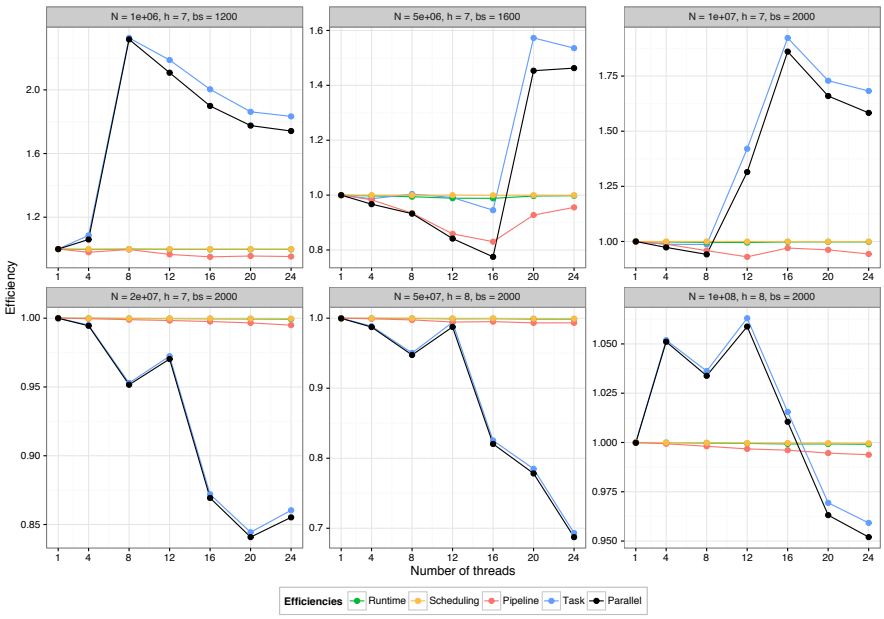



Figure 29: efficiencies for cube (volume) with Klang-for-dyn

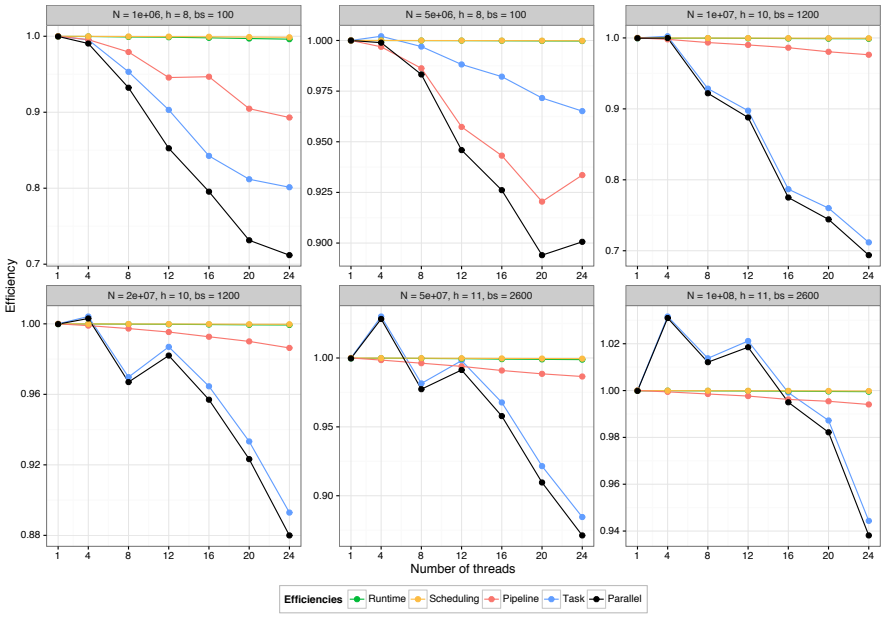



Figure 30: efficiencies for ellipsoid (surface) with Klang-for-dyn

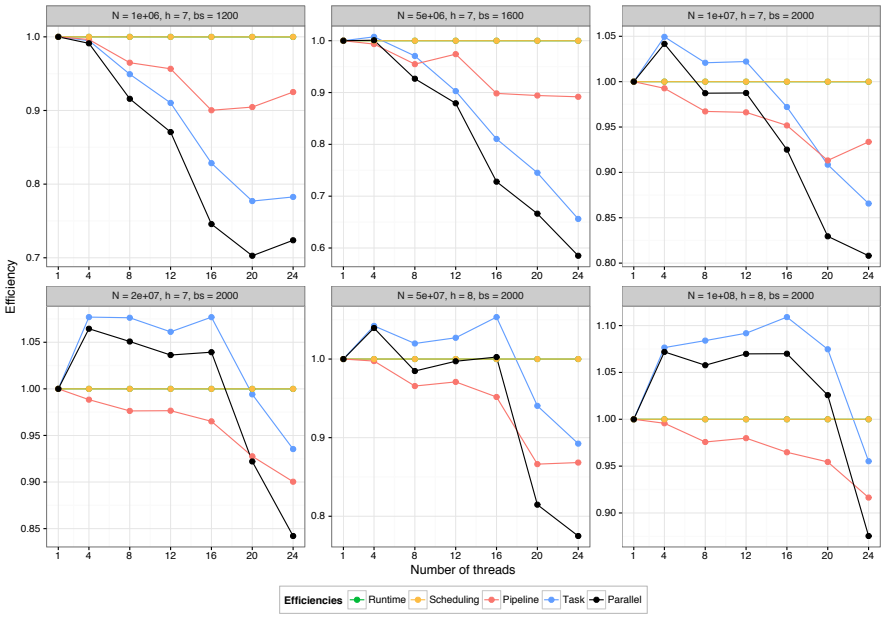



Figure 31: efficiencies for cube (volume) with Klang-for-stat

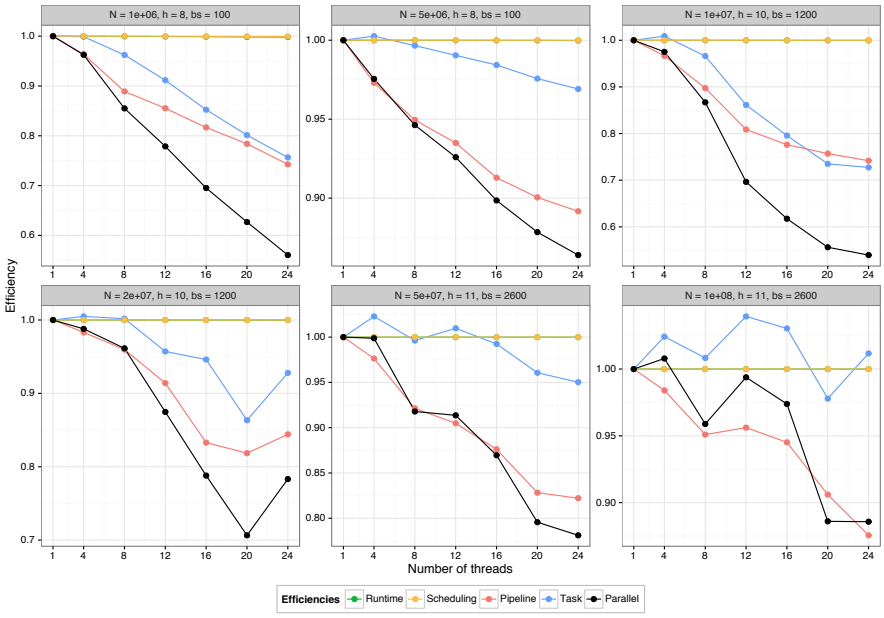



Figure 32: efficiencies for ellipsoid (surface) with Klang-for-stat

### A.1.6 Accuracy of detailed timings

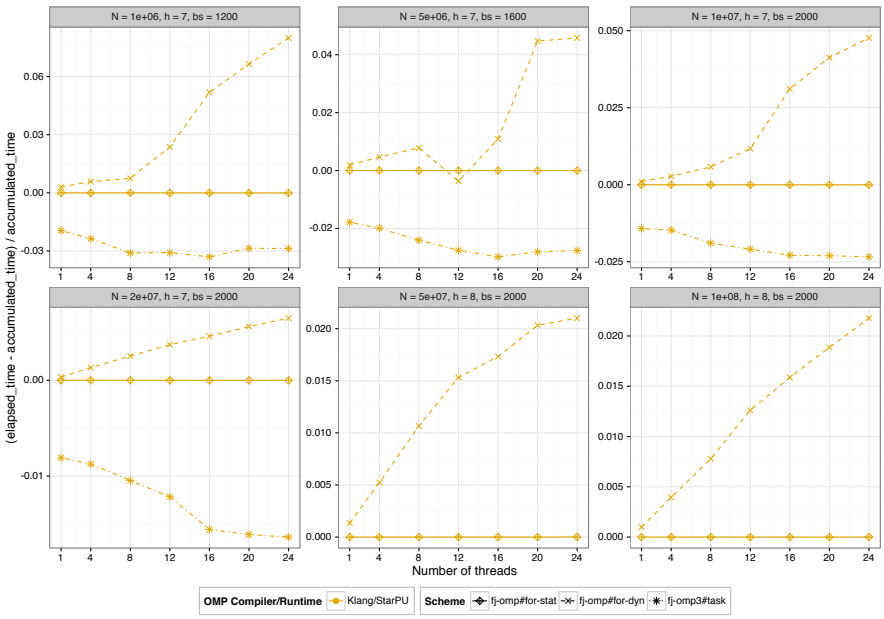



Figure 33: errors for cube (volume)

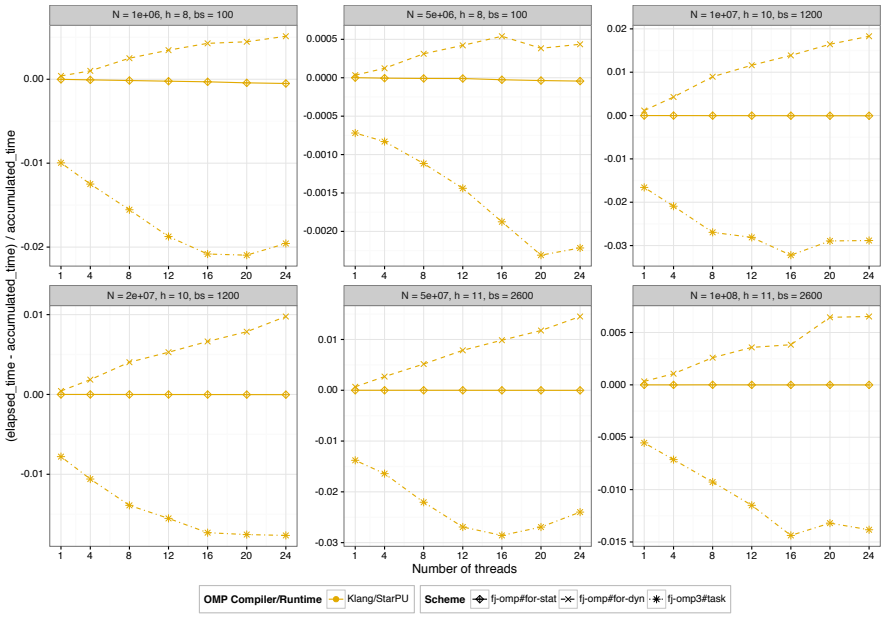



Figure 34: errors for ellipsoid (surface)

## A.2 Performance of task-based schemes

### A.2.1 Normalized efficiencies

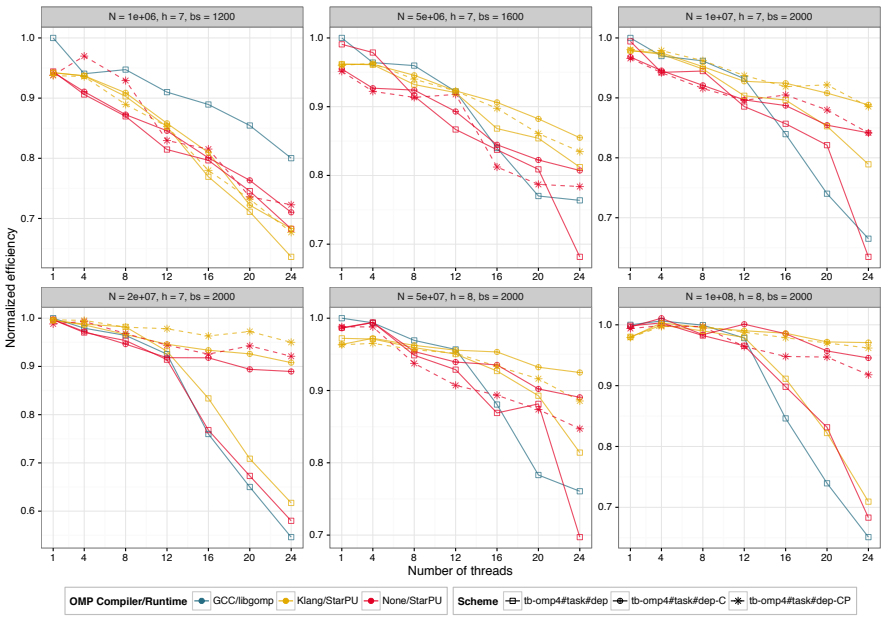



Figure 35: Normalized efficiencies for cube (volume)

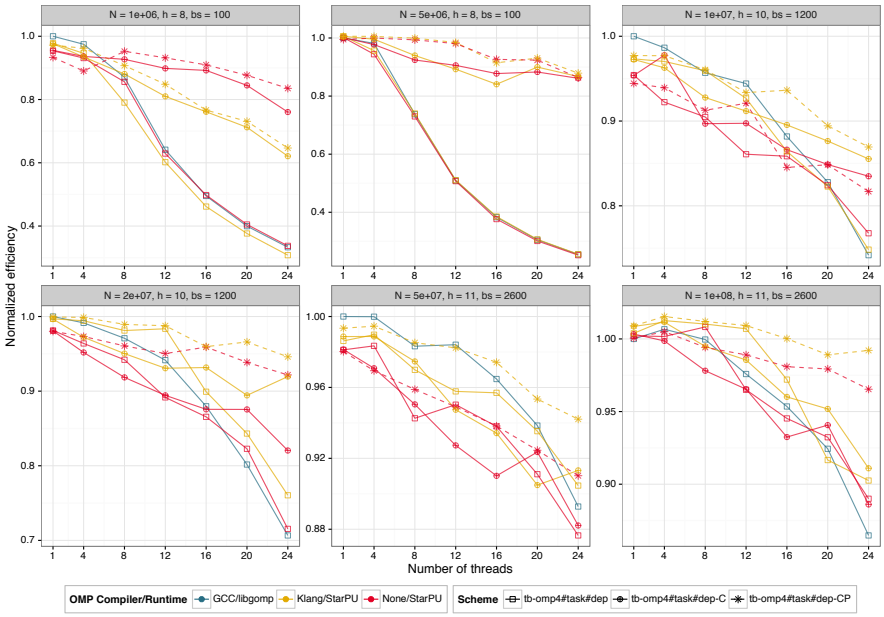



Figure 36: Normalized efficiencies for ellipsoid (surface)

### A.2.2 Speedup

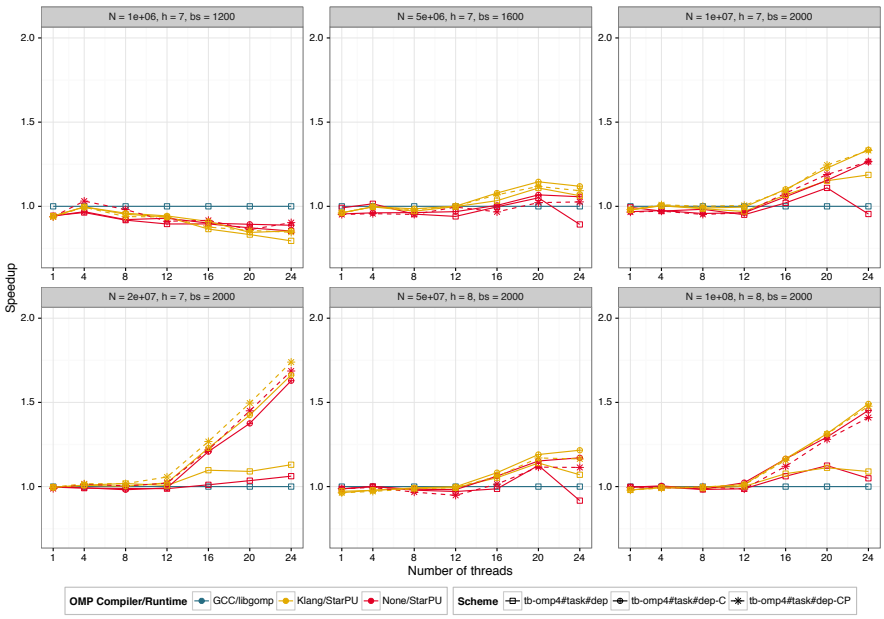



Figure 37: Speedup for cube (volume)

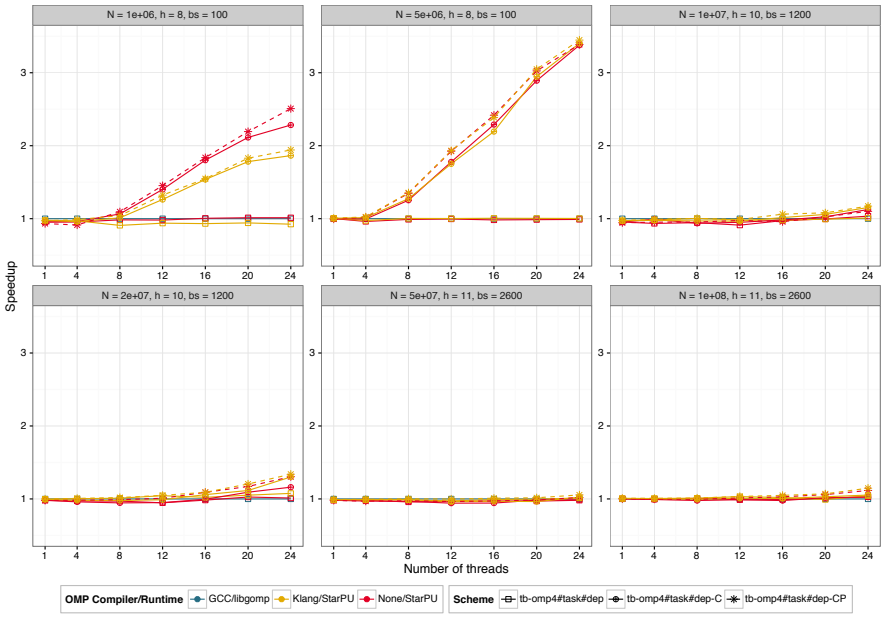



Figure 38: Speedup for ellipsoid (surface)

### A.2.3 Parallel efficiency

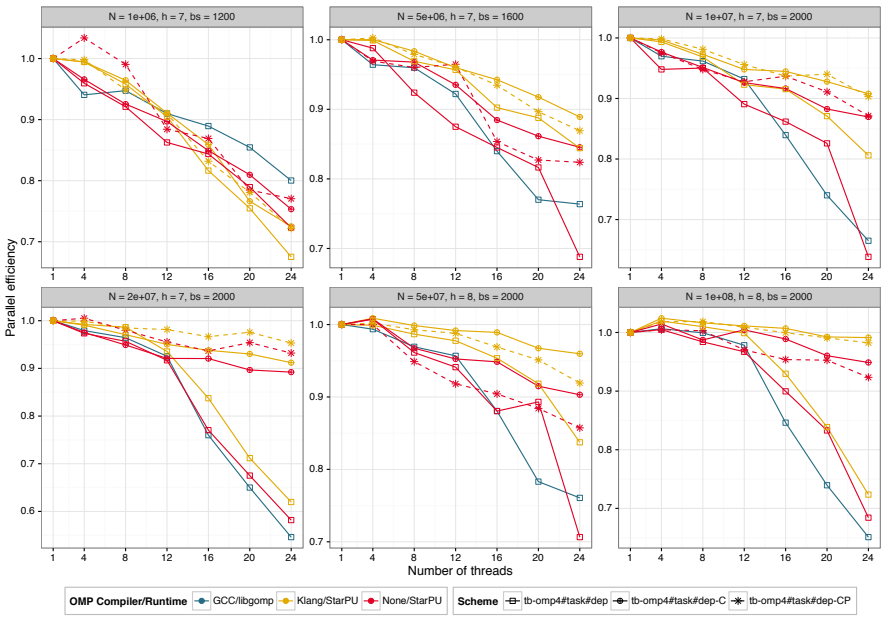



Figure 39: Parallel efficiency for cube (volume)

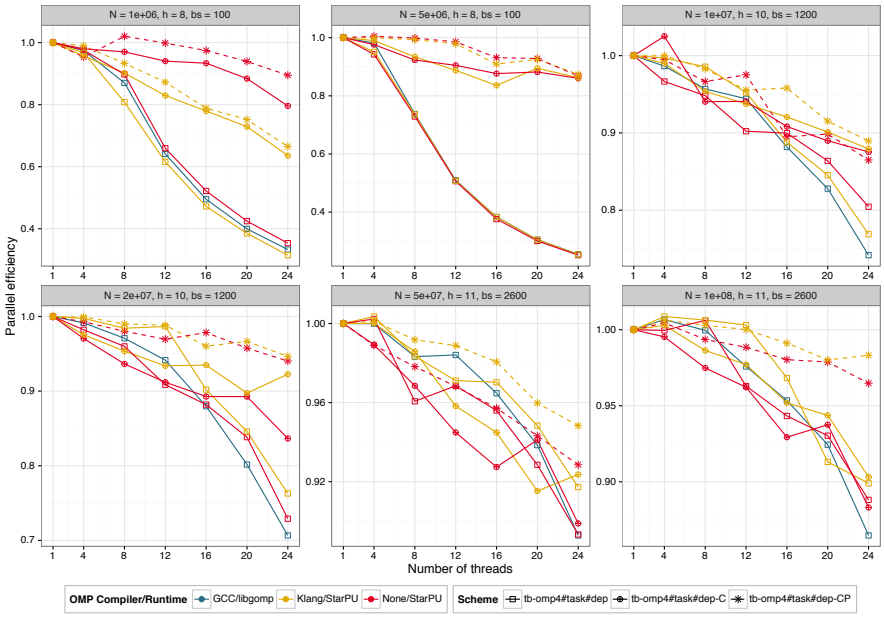



Figure 40: Parallel efficiency for ellipsoid (surface)

### A.2.4 Timings

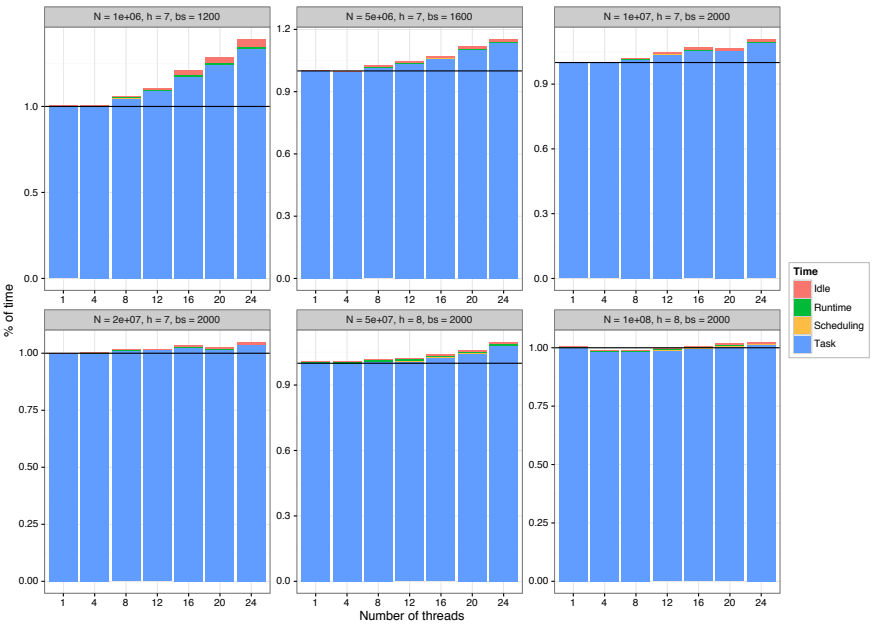



Figure 41: timings for cube (volume) with Klang-CP

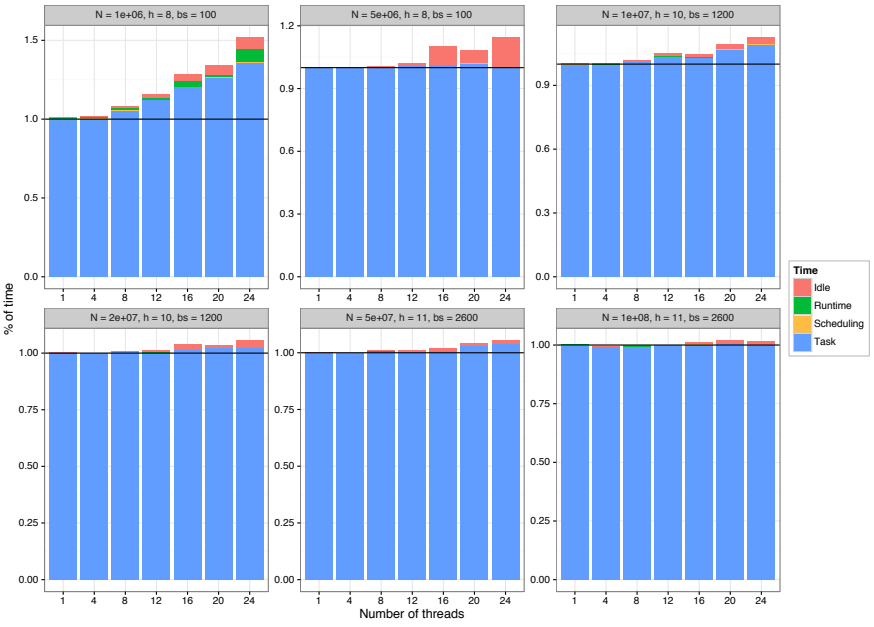



Figure 42: timings for ellipsoid (surface) with Klang-CP

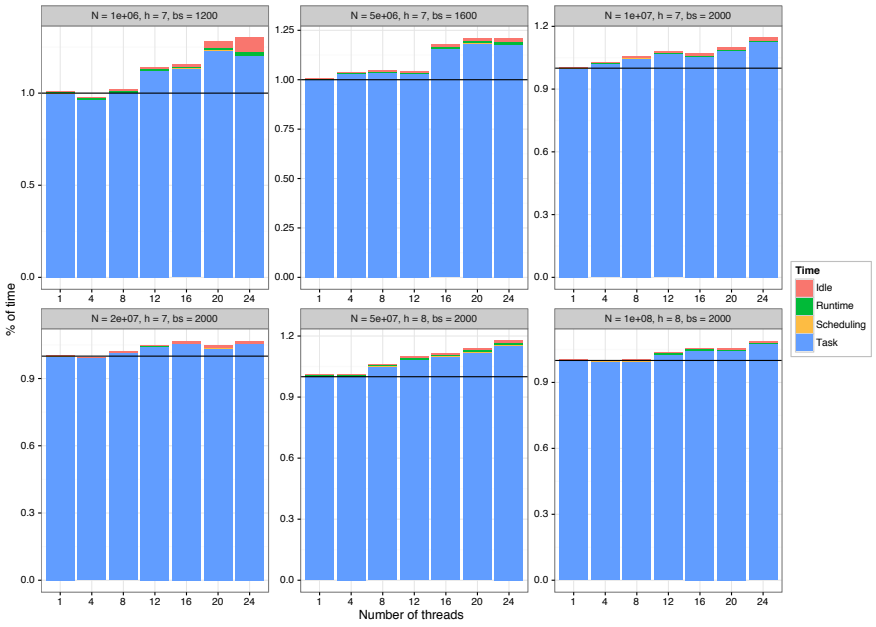



Figure 43: timings for cube (volume) with StarPU-CP

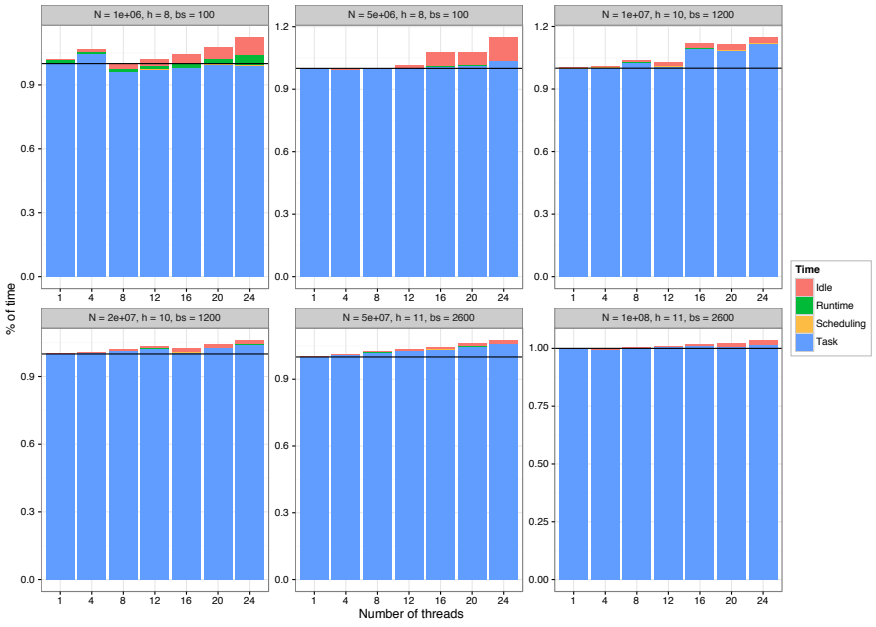



Figure 44: timings for ellipsoid (surface) with StarPU-CP

### A.2.5 Efficiencies

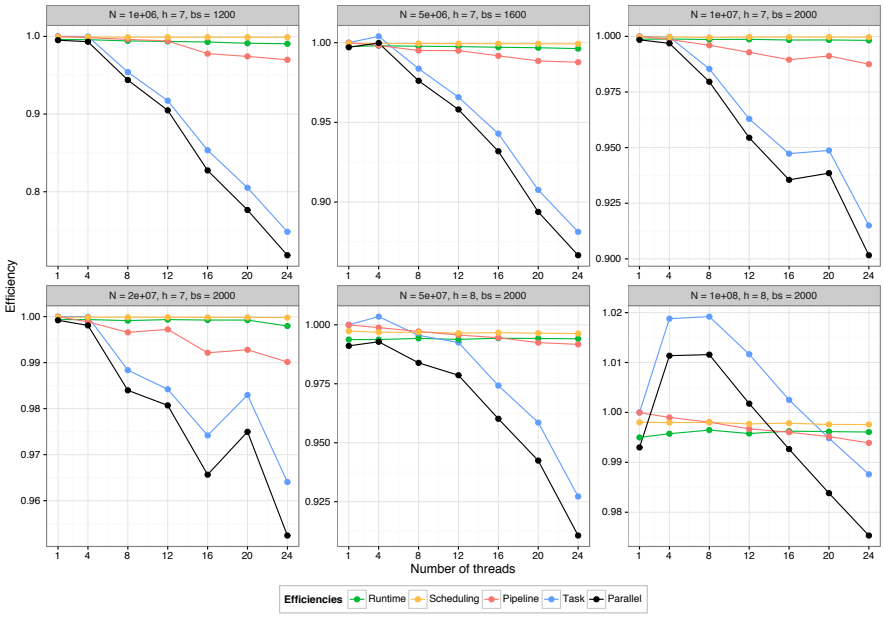



Figure 45: efficiencies for cube (volume) with Klang-CP

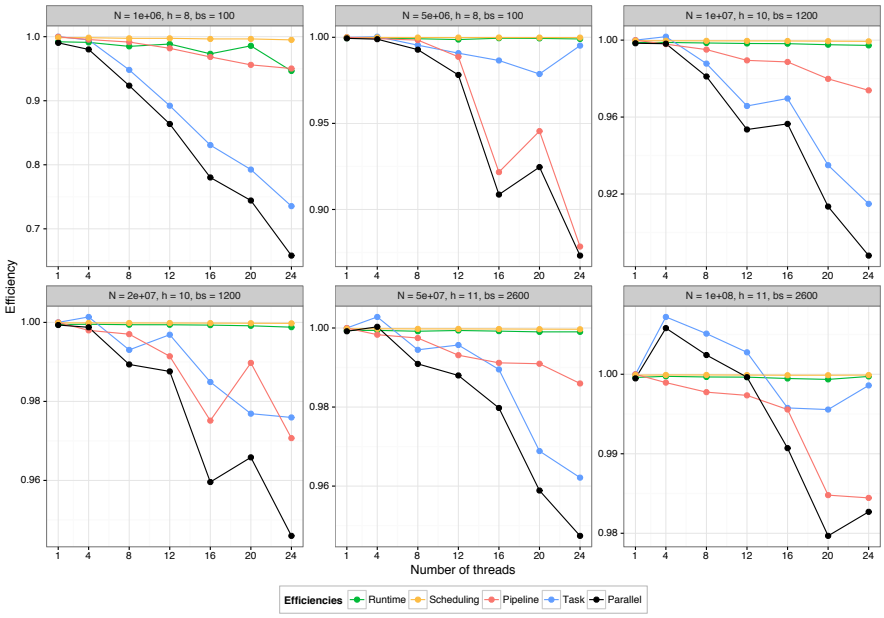



Figure 46: efficiencies for ellipsoid (surface) with Klang-CP

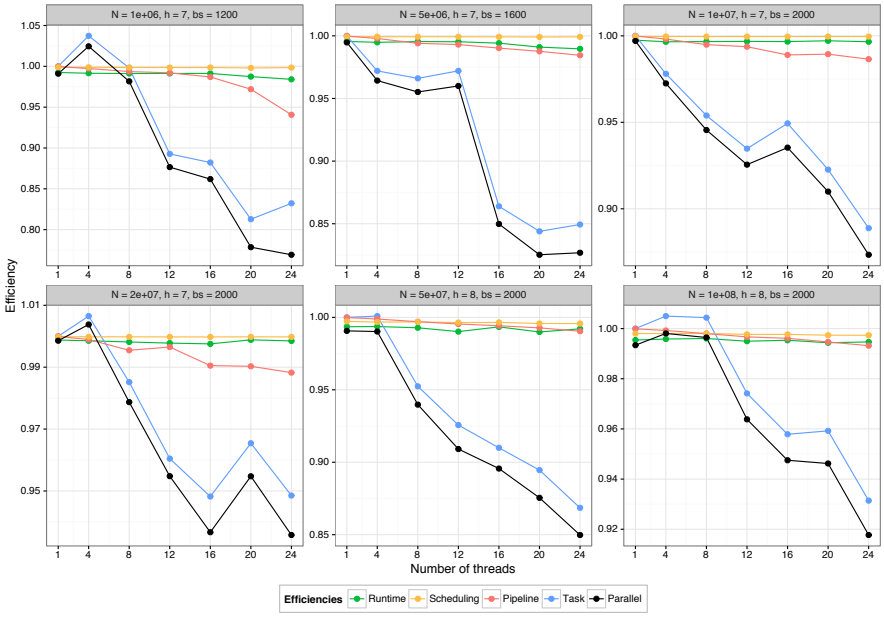



Figure 47: efficiencies for cube (volume) with StarPU-CP

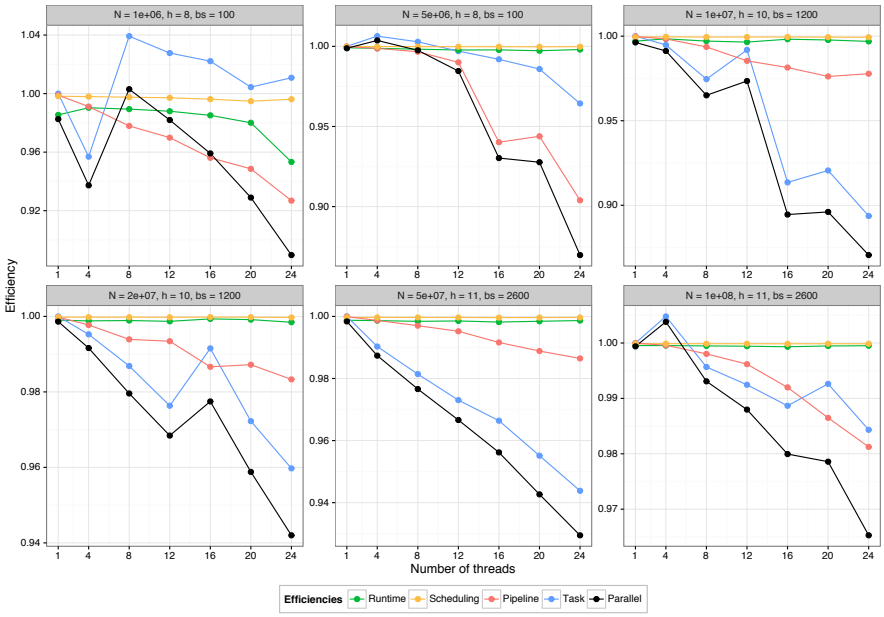



Figure 48: efficiencies for ellipsoid (surface) with StarPU-CP

### A.2.6 Accuracy of detailed timings

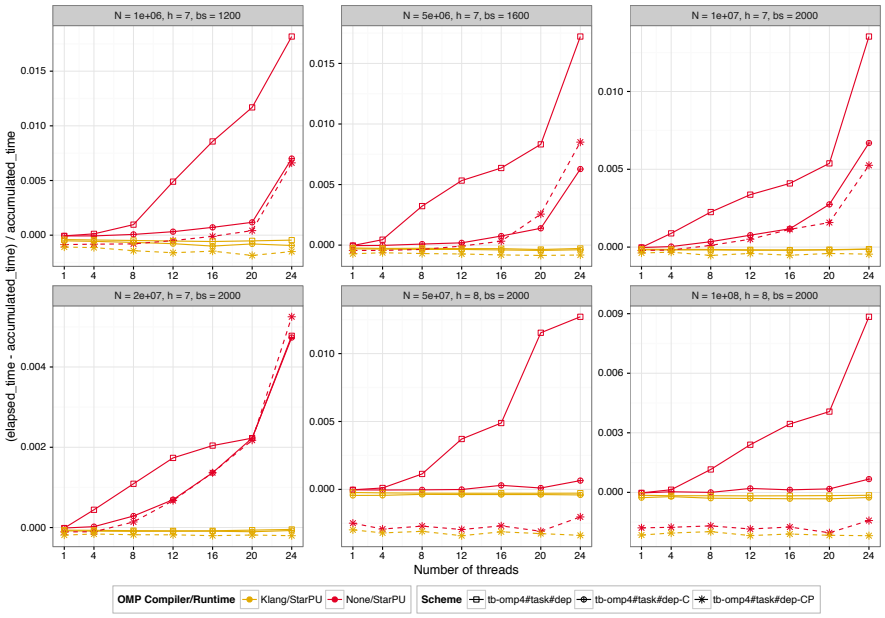



Figure 49: errors for cube (volume)

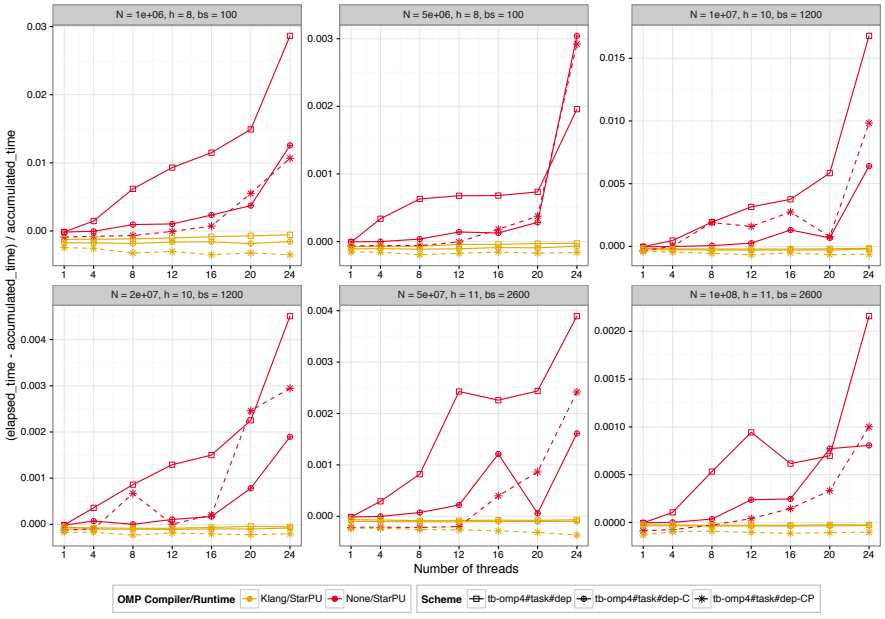



Figure 50: errors for ellipsoid (surface)

# References

[1] *Diagnosis, Tuning, and Redesign for Multicore Performance: A Case Study of the Fast Multipole Method*, in Proceedings of the 2010 ACM/IEEE conference on Supercomputing, 2010.

[2] *Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures*, in 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), IEEE, 2010, pp. 1–12, http://dx.doi.org/10.1109/IPDPS.2010.5470415, http://www.cc.gatech.edu/{~}gbiros/http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5470415.

[3] *LLVM OpenMP Runtime Library*, 2015, http://openmp.llvm.org/Reference.pdf/.

[4] C. Addisson, J. Lagrone, L. Huang, and B. Chapman, *OpenMP 3.0 tasking implementation in OpenUH*, in Open64 Workshop, 2009.

[5] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, *Task-based FMM for multicore architectures*, SIAM Journal on Scientific Computing, 36 (2014), pp. C66–C93, http://dx.doi.org/10.1137/130915662.

[6] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, *Task-based FMM for heterogeneous architectures*, Concurrency and Computation: Practice and Experience, 28 (2016), pp. 2608–2629, http://dx.doi.org/10.1002/cpe.3723.

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, *StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures*, Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, (2011).

[8] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, *The design of openmp tasks*, Transactions on Parallel and Distributed Systems, (2009).

[9] P. Blanchard, O. Coulaud, and E. Darve, *Fast hierarchical algorithms for generating Gaussian random fields*, Research Report 8811, Inria Bordeaux Sud-Ouest, Dec. 2015, https://hal.inria.fr/hal-01228519.

[10] G. Bosilca, A. Bouteiller, A. Danalis, T. Hérault, P. Lemarinier, and J. Dongarra, *Dague: A generic distributed dag engine for high performance computing*, Parallel Computing, (2012).

[11] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, *Implementing ompss support for regions of data in architectures with multiple address spaces*, in International conference on Supercomputing, 2013.

[12] H. Casanova, A. Legrand, and Y. Robert, *Parallel algorithms*, CRC Press, 2008.

[13] J. Choi, A. Chandramowlishwaran, K. Madduri, and R. Vuduc, *A cpu: Gpu hybrid implementation and model-driven scheduling of the fast multipole method*, in Proceedings of Workshop on General Purpose Processing Using GPUs, GPGPU-7, New York, NY, USA, 2014, ACM, pp. 64:64–64:71, http://doi.acm.org/10.1145/2576779.2576787.

[14] R. Ferrer, S. Royuela, D. Caballero, A. Duran, X. Martorell, and E. Ayguadé, *Mercurium: Design decisions for a s2s compiler*, Cetus Users and Compiler Infastructure Workshop, (2011).

[15] M. Frigo, C. E. Leiserson, and K. H. Randall, *The implementation of the cilk-5 multithreaded language*, in Conference on Programming Language Design and Implementation, 1998.

[16] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, *Locality-aware work stealing on multi-cpu and multi-gpu architectures*, in Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG), 2013.

[17] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, Journal of computational physics, 73 (1987), pp. 325–348.

[18] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski, *A rose-based openmp 3.0 research compiler supporting multiple runtime libraries*, in Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, Springer, 2010.

[19] H. Ltaief and R. Yokota, *Data-driven execution of fast multipole methods*, Concurrency and Computation: Practice and Experience, (2013), pp. 1935–1946.

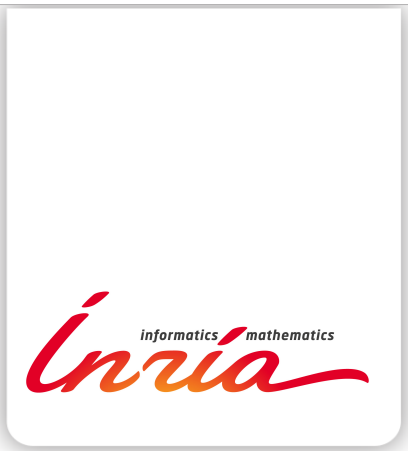
[20] Open Source Technology Center, *Open Community Runtime*, 2014, `https://01.org/open-community-runtime/`.

[21] OpenMP Architecture Review Board, *OpenMP application program interface version 4.0*, 2013, `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[22] OpenMP Architecture Review Board, *OpenMP application program interface version 4.5*, 2015, `http://www.openmp.org/mp-documents/openmp-4.5.pdf`.

[23] X. Teruel, P. Unnikrishnan, X. Martorell, E. Ayguadé, R. Silvera, G. Zhang, and E. Tiotto, *Openmp tasks in IBM XL compilers*, in Conference of the center for advanced studies on collaborative research, 2008.

[24] R. Yokota and L. A. Barba, *A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems*, International Journal of High Performance Computing Applications, 26 (2012), pp. 337–346.




**RESEARCH CENTRE**
**BORDEAUX – SUD-OUEST**

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399